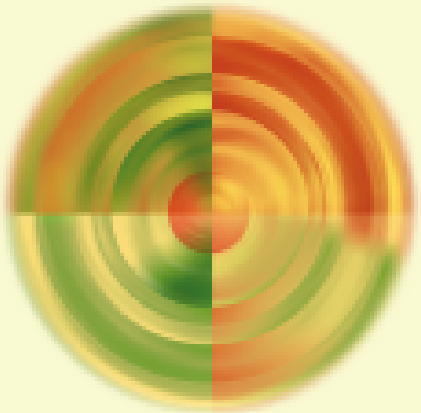
GPU Programming

(from Python)

Paul Ivanov

UC Berkeley
Vision Science Graduate Program



Redwood Center for Theoretical Neuroscience
`http://redwood.berkeley.edu/`

October 29, 2012

An illustrative analogy: candy eating contest

flickr/stallio “classic jack-o-lantern”

Who would win: Fat bastard, or . . .

credit: Austin Powers (New Line Cinema)

. . . a toddler?

FB vs toddler: no contest

vs

FB vs many toddlers

vs


vs


So who would win?

The answer is: it depends on how much candy there is to eat!

if it's just one bucket - FB wins

if it's a truck load, the kids win!

CPU vs GPU

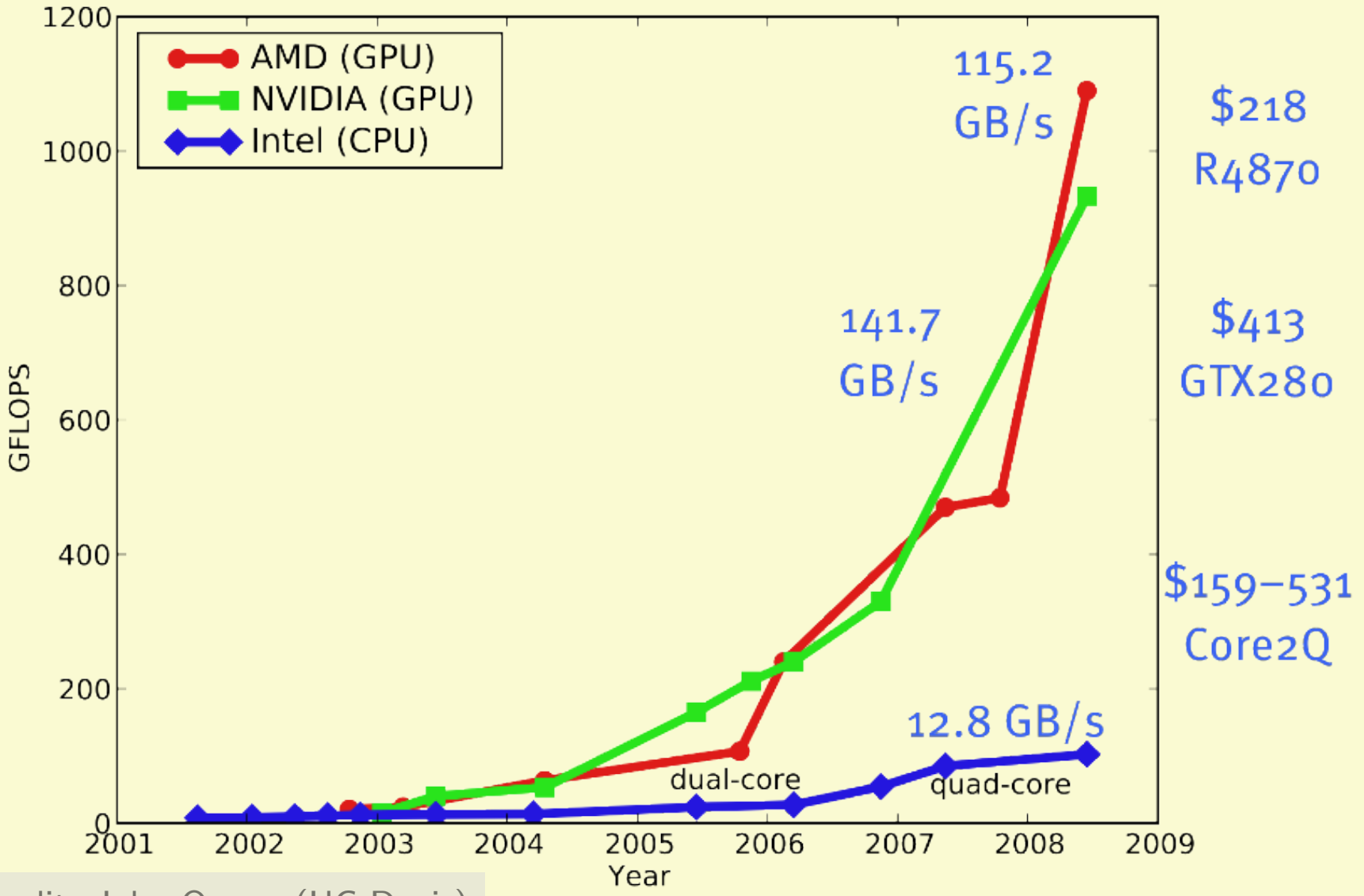

vs


CPU vs GPU

The analogy is the same, and even many nuances translate. . .
Kids are more naive, less sophisticated
They aren't as fast, either, but their power is in numbers
Kids need to be supervised and directed (work best in unison)

Why GPUs?

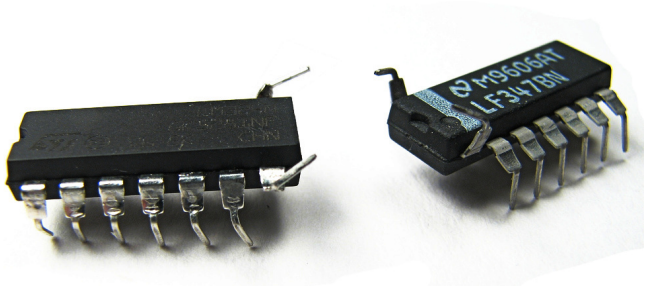
Programmable 32-bit FP operations per second



credit: John Owens (UC Davis)

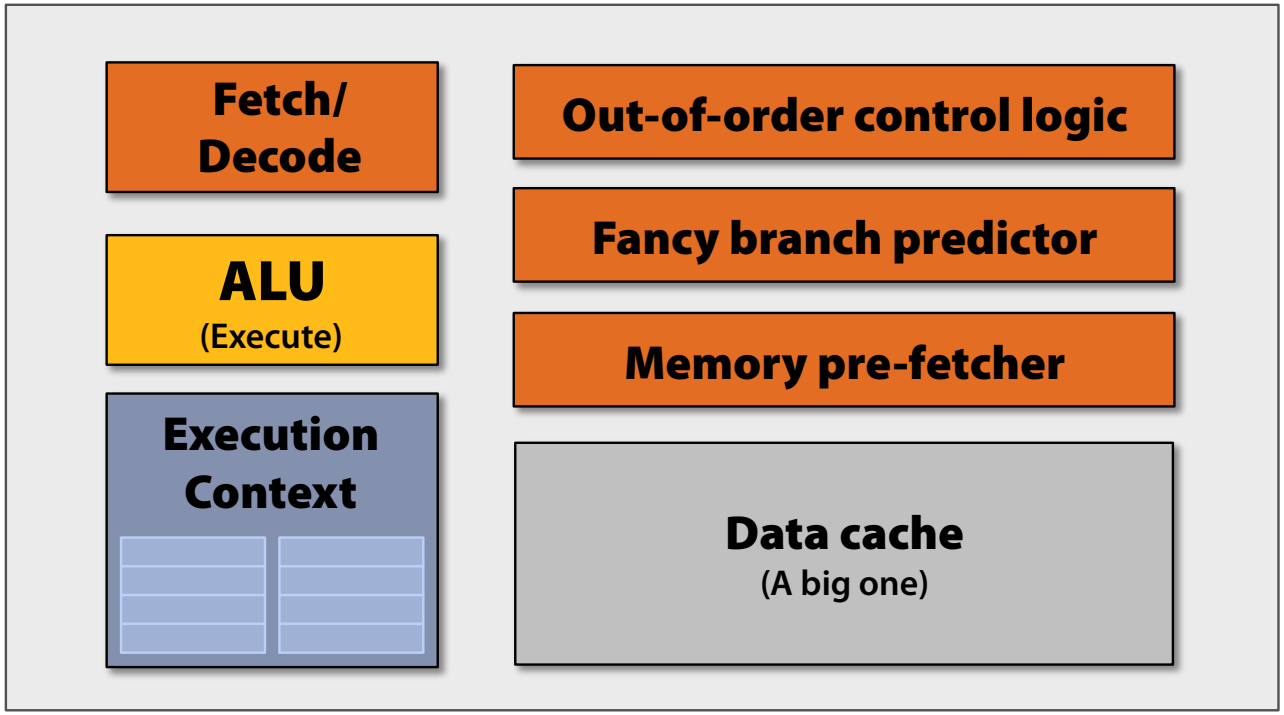
GPU Computing?

- Design target for CPUs:
  - Make a single thread very fast
  - Take control away from programmer
- GPU Computing takes a different approach:
  - Throughput matters—single threads do not
  - Give explicit control to programmer



credit: Andreas Klöckner (NYU)

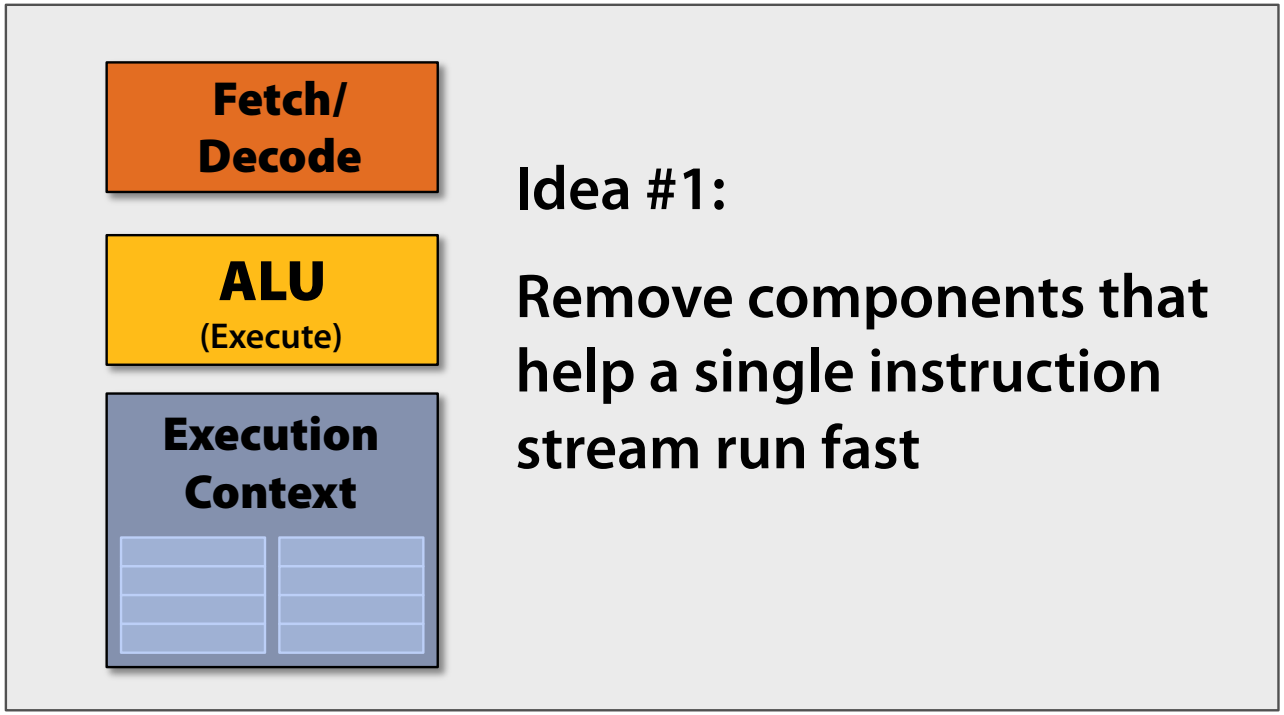
"CPU-style" Cores



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

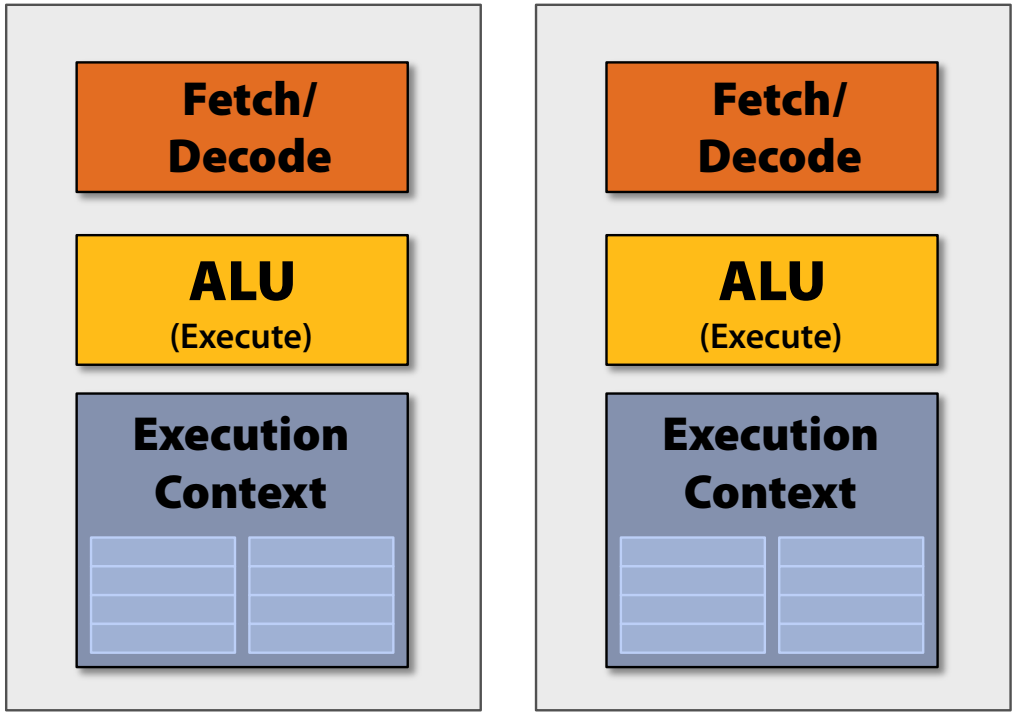
Slimming down



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

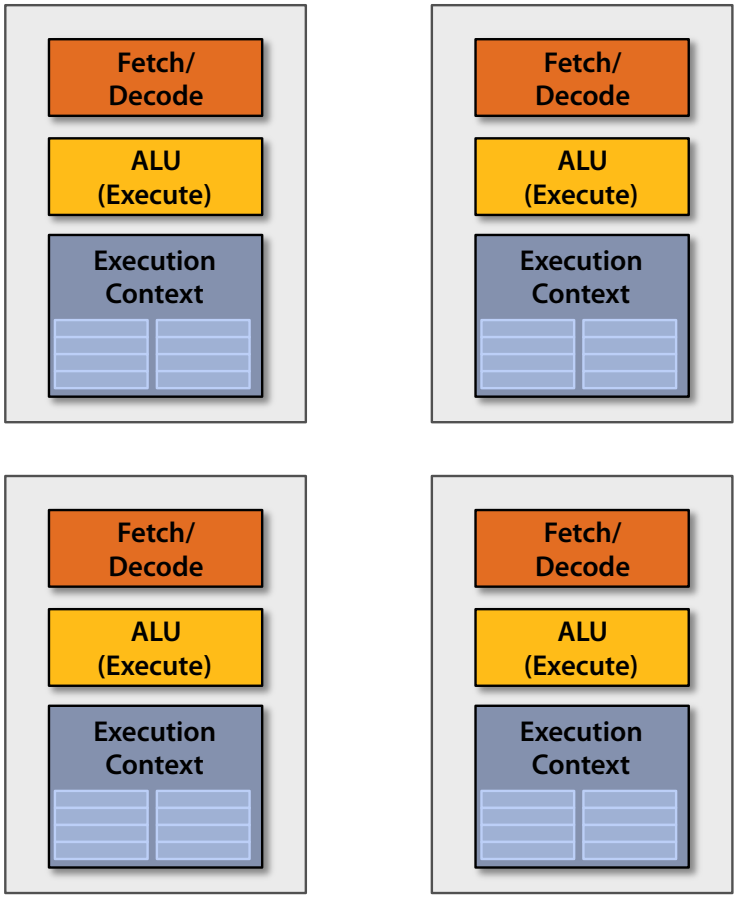
More Space: Double the Number of Cores



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

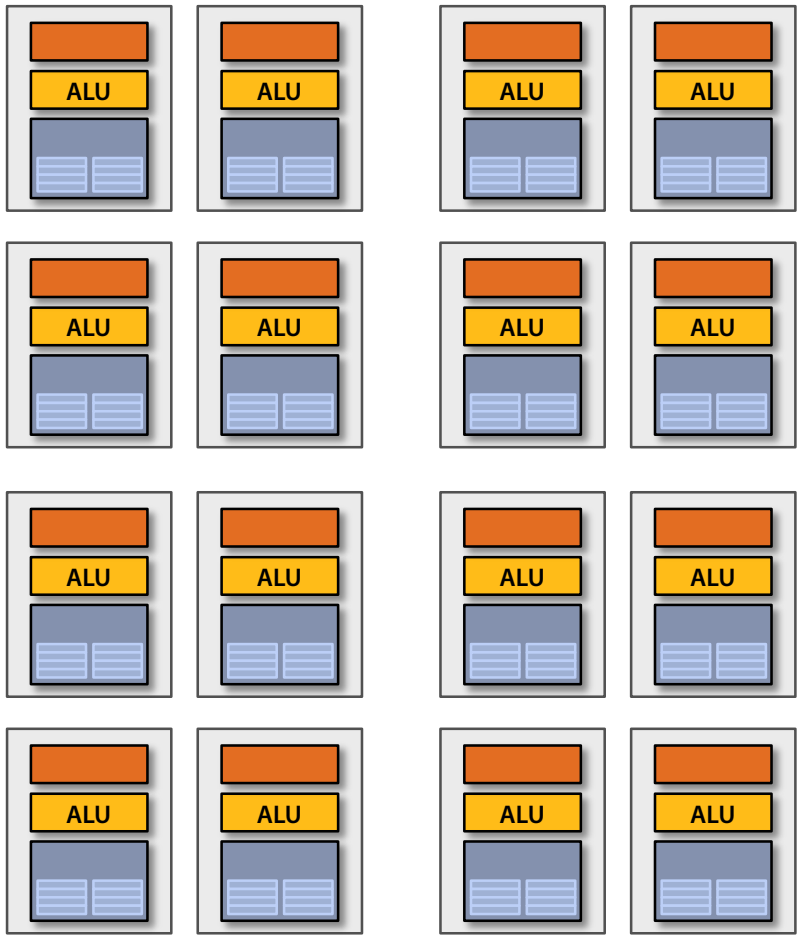
. . . again



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

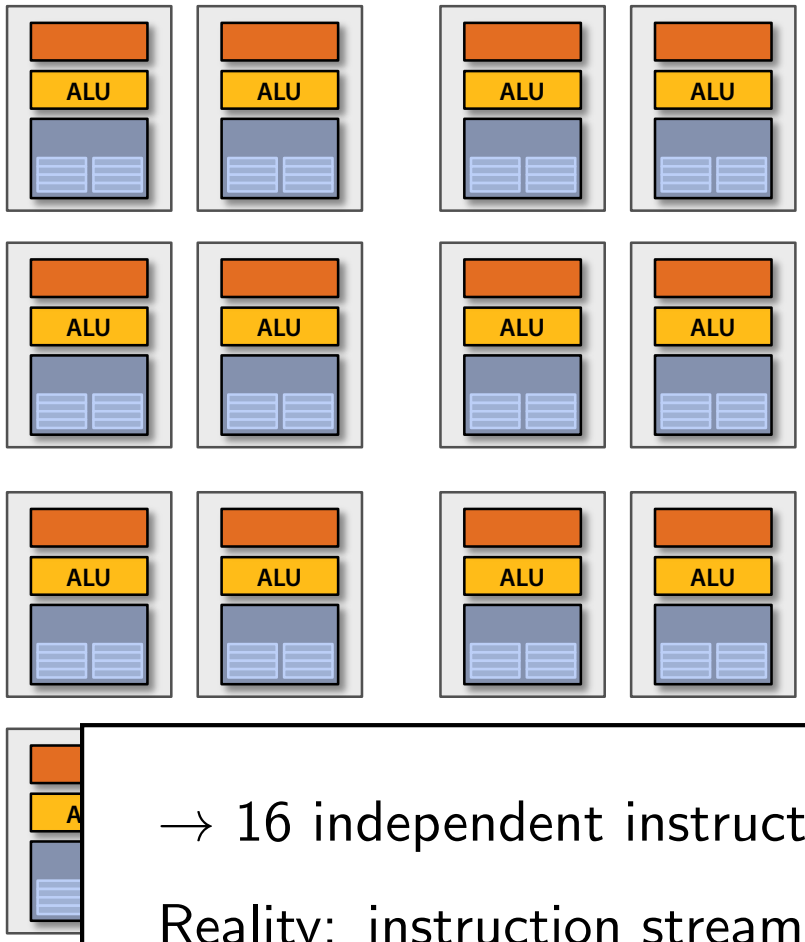
. . . and again



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

. . . and again



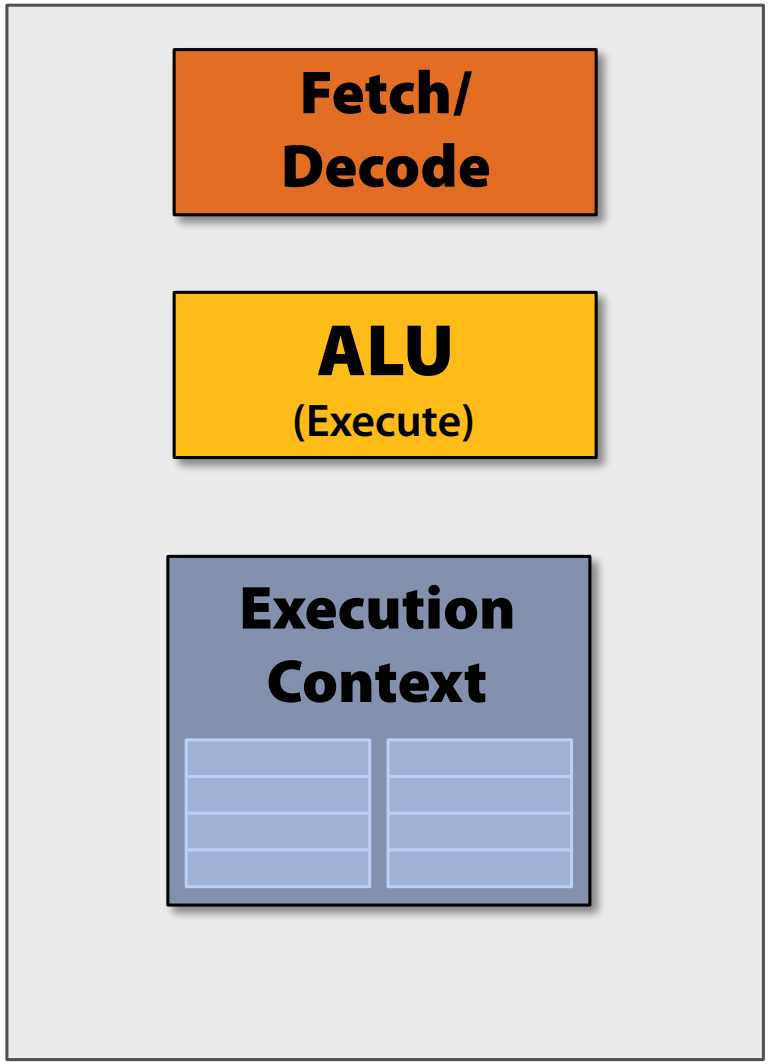
→ 16 independent instruction streams

Reality: instruction streams not actually very different/independent

Credit: Kayvon Fatahalian (

credit: Andreas Klöckner (NYU)

Saving Yet More Space



Credit: Kayvon Fatahalian (Stanford)

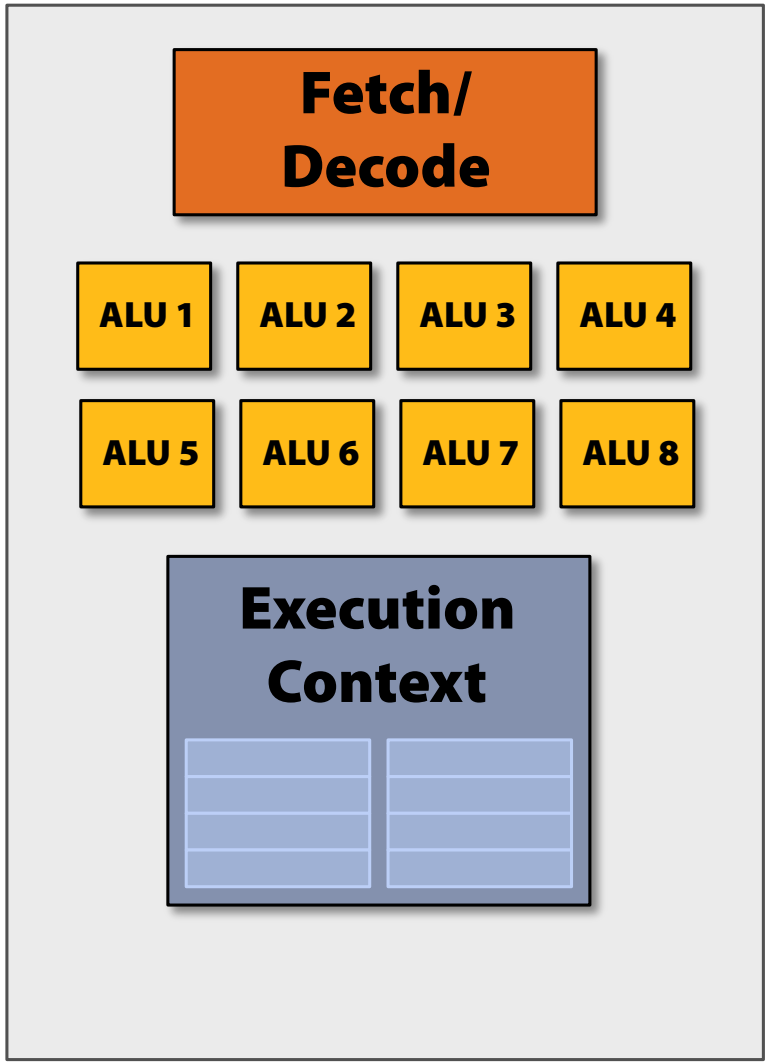
Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

→ **SIMD**

credit: Andreas Klöckner (NYU)

Saving Yet More Space



Credit: Kayvon Fatahalian (Stanford)

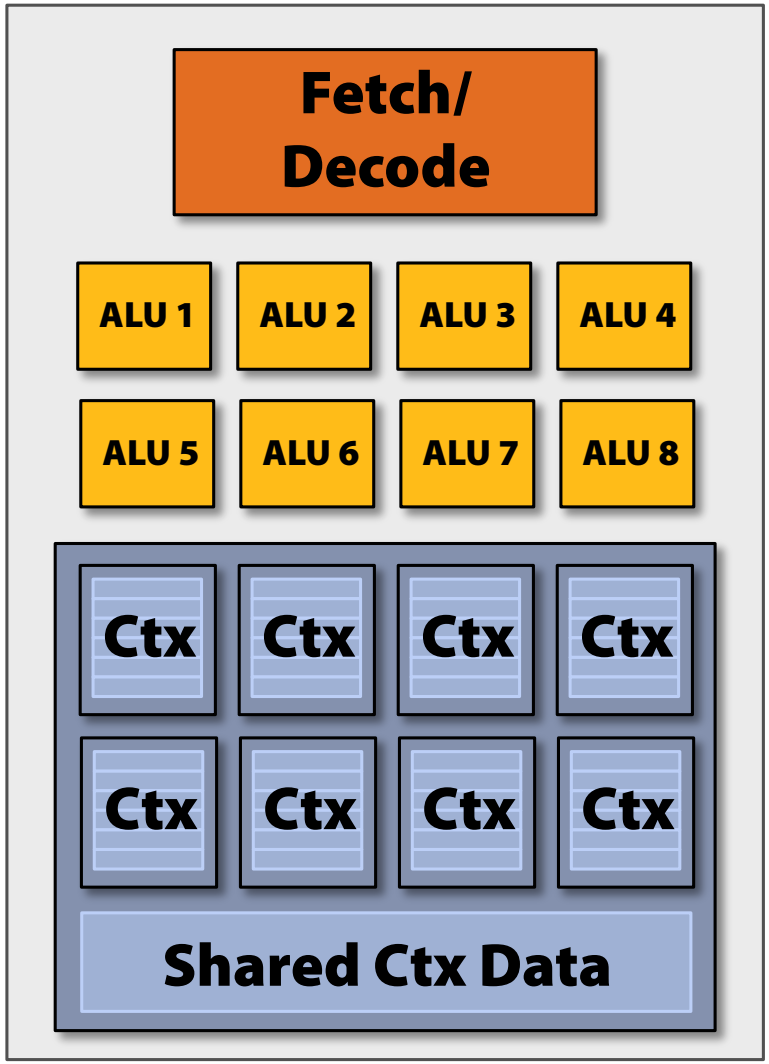
Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

$\rightarrow$ **SIMD**

credit: Andreas Klöckner (NYU)

Saving Yet More Space



Credit: Kayvon Fatahalian (Stanford)

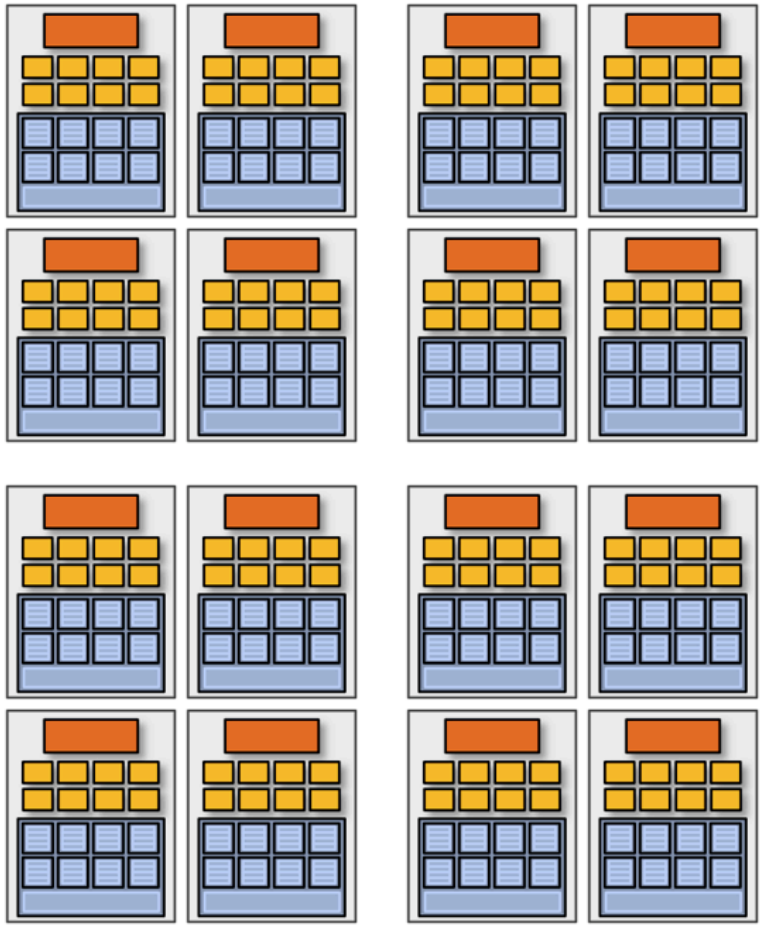
Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

$\rightarrow$ **SIMD**

credit: Andreas Klöckner (NYU)

Gratuitous Amounts of Parallelism!



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

Gratuitous Amounts of Parallelism!

Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams

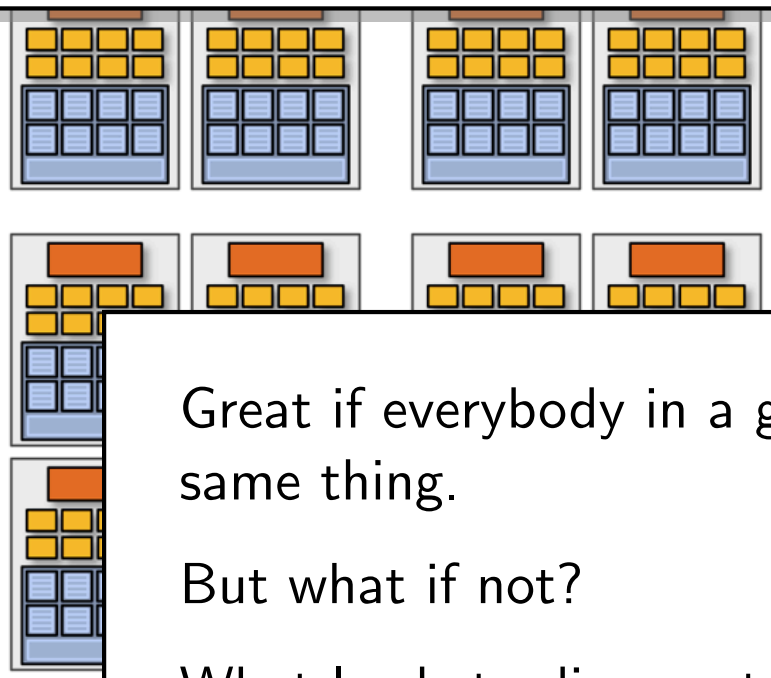
Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

Gratuitous Amounts of Parallelism!

Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams



Great if everybody in a group does the same thing.

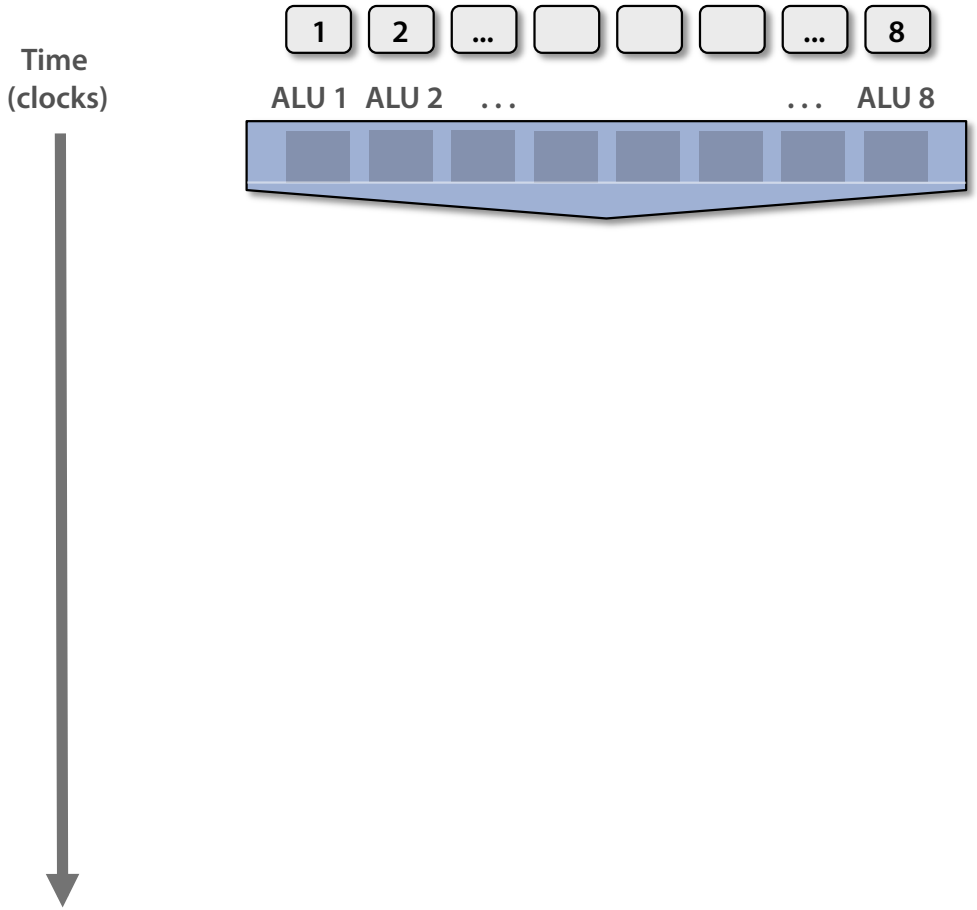
But what if not?

What leads to divergent instruction streams?

Credit: Kayvon Fatahalian (

credit: Andreas Klöckner (NYU)

Branches

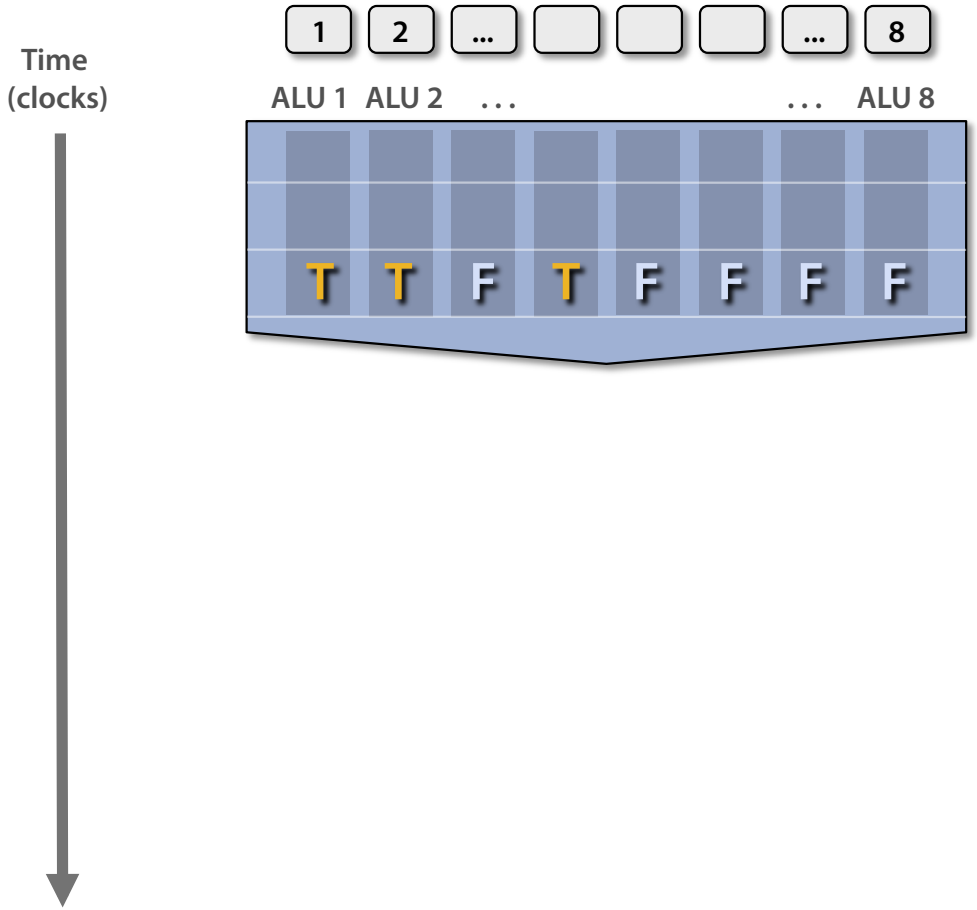



```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

Branches

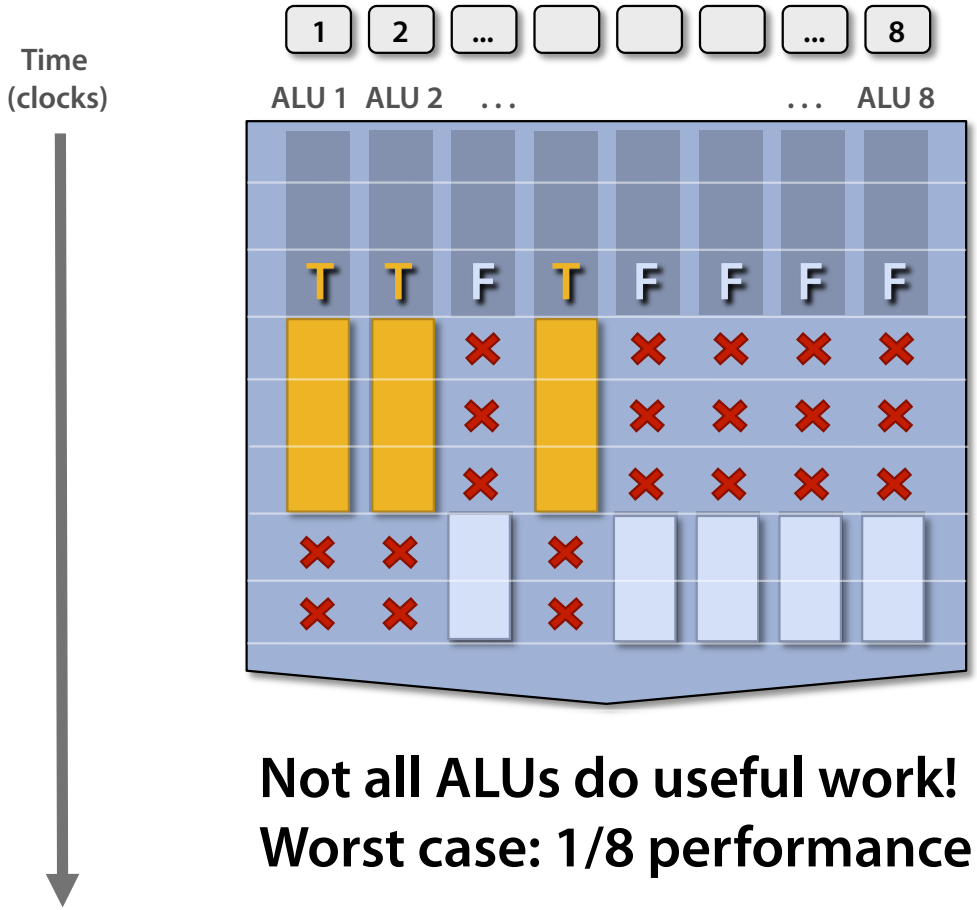



```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

Branches



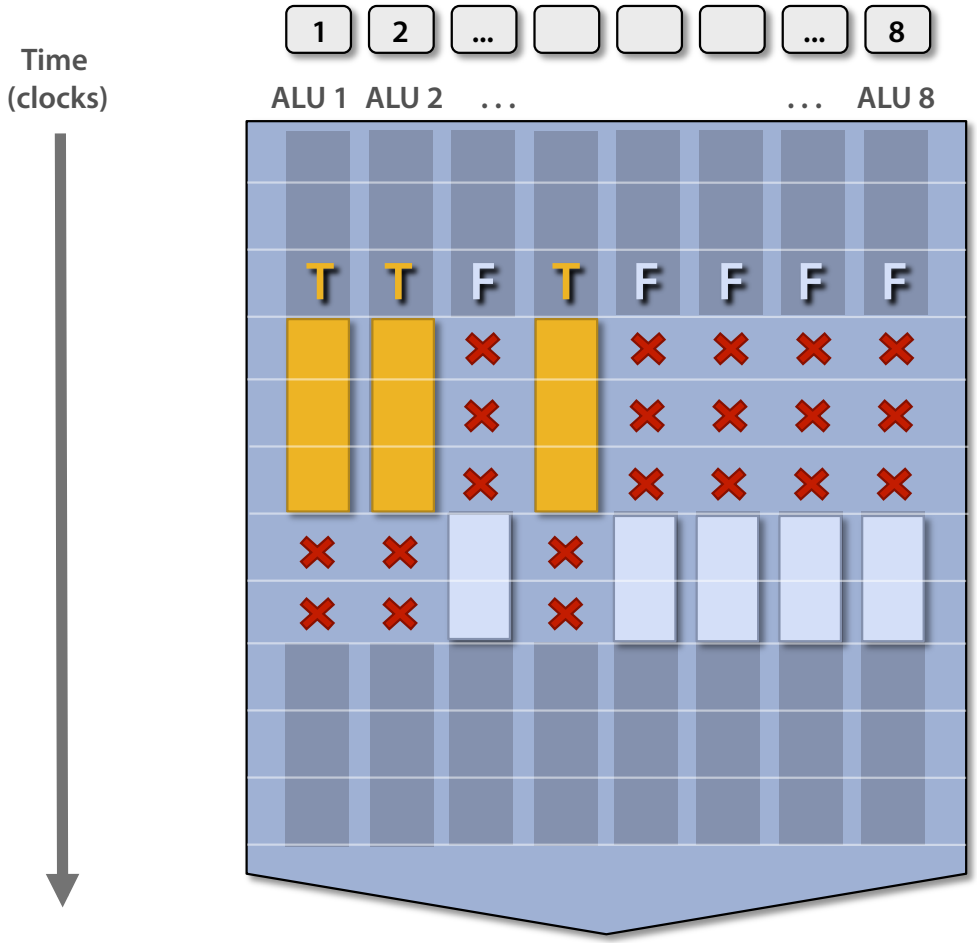

```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Not all ALUs do useful work!
Worst case: 1/8 performance

Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

Branches




```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

Remaining Problem: Slow Memory

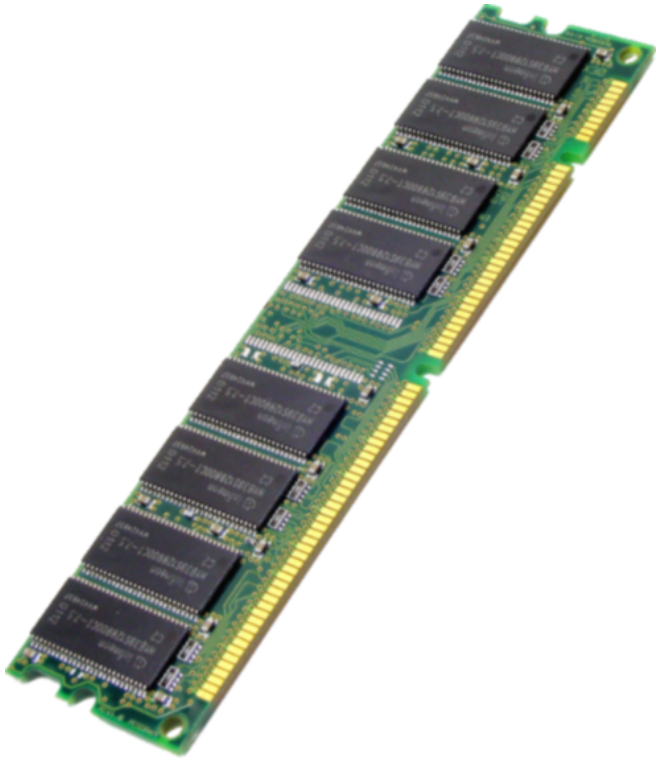
Problem

Memory still has very high latency...
...but we've removed most of the hardware that helps us deal with that.

We've removed

- caches
- branch prediction
- out-of-order execution

So what now?



credit: Andreas Klöckner (NYU)

Remaining Problem: Slow Memory

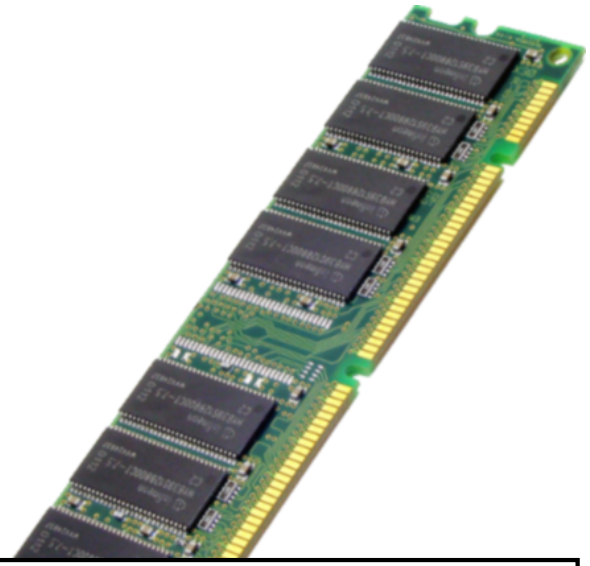
Problem

Memory still has very high latency...
...but we've removed most of the hardware that helps us deal with that.

We've removed

- caches
- branch prediction
- out-of-order execution

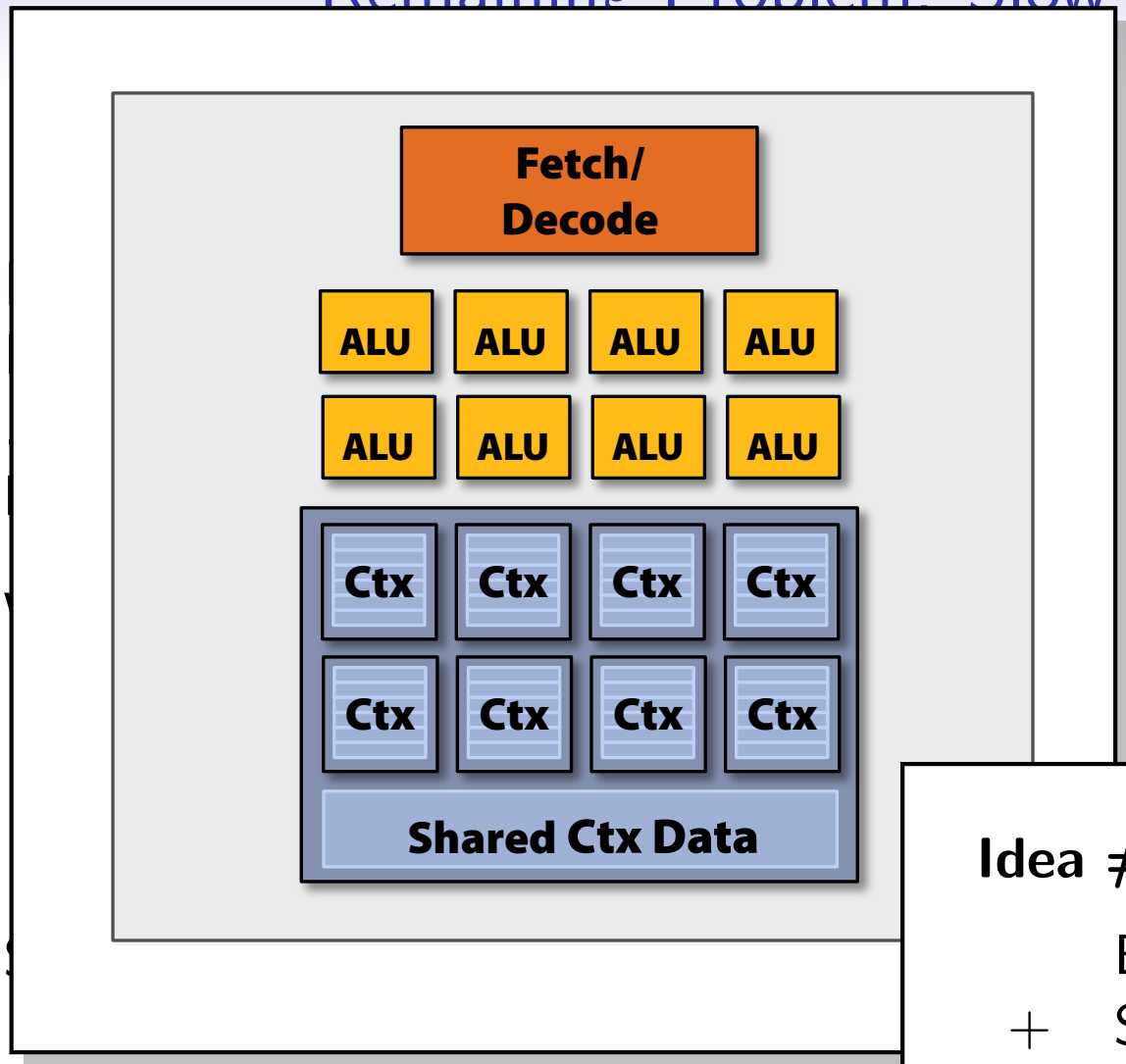
So what now?

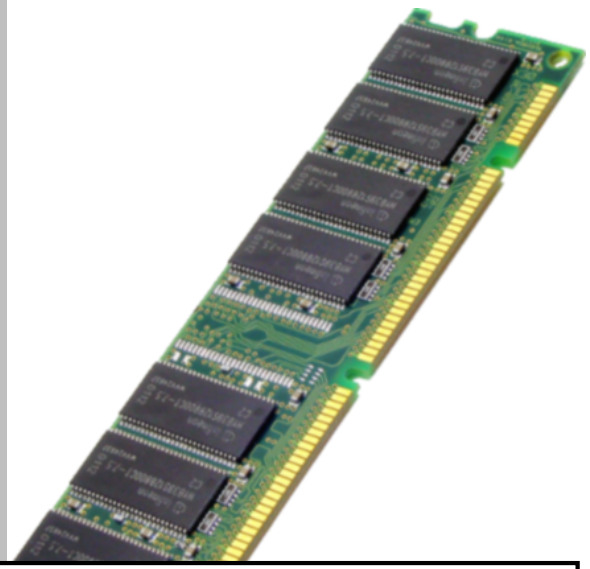


Idea #3

	Even more parallelism
+	Some extra memory
=	A solution!

Remaining Problem: Slow Memory



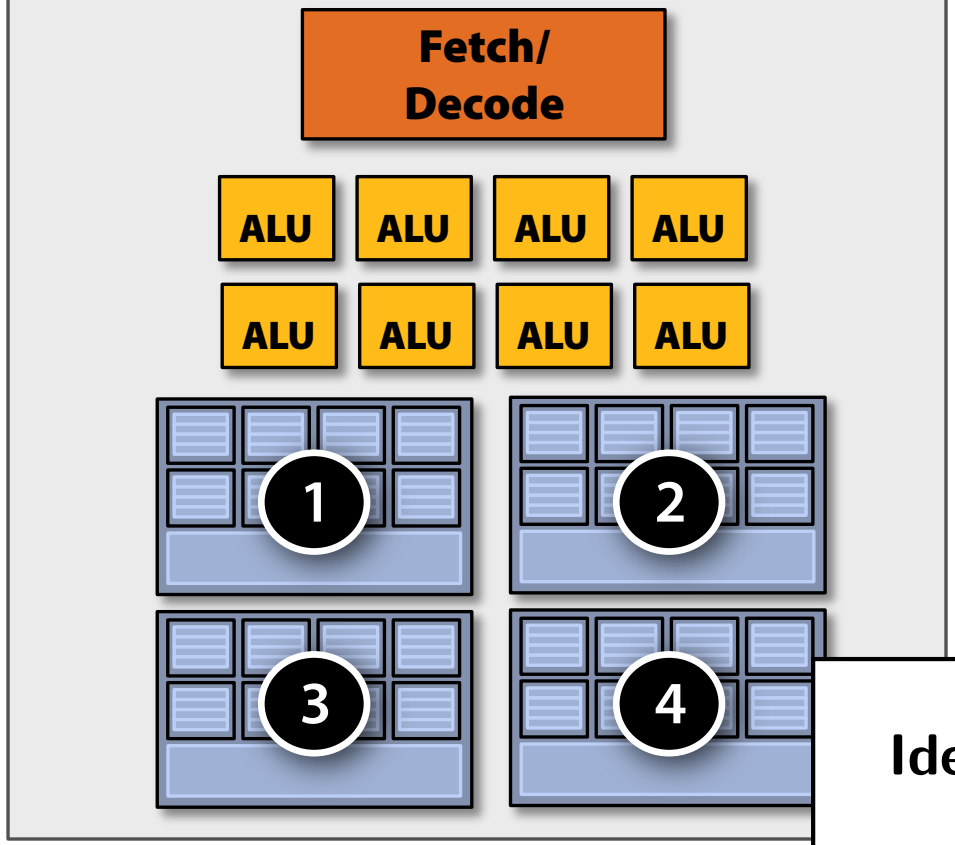


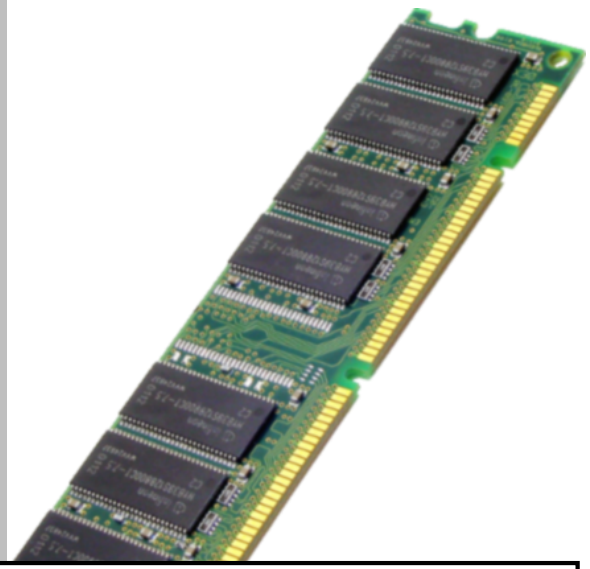
Idea #3

	Even more parallelism
+	Some extra memory
=	A solution!

credit: Andreas Klöckner (NYU)

Remaining Problem: Slow Memory



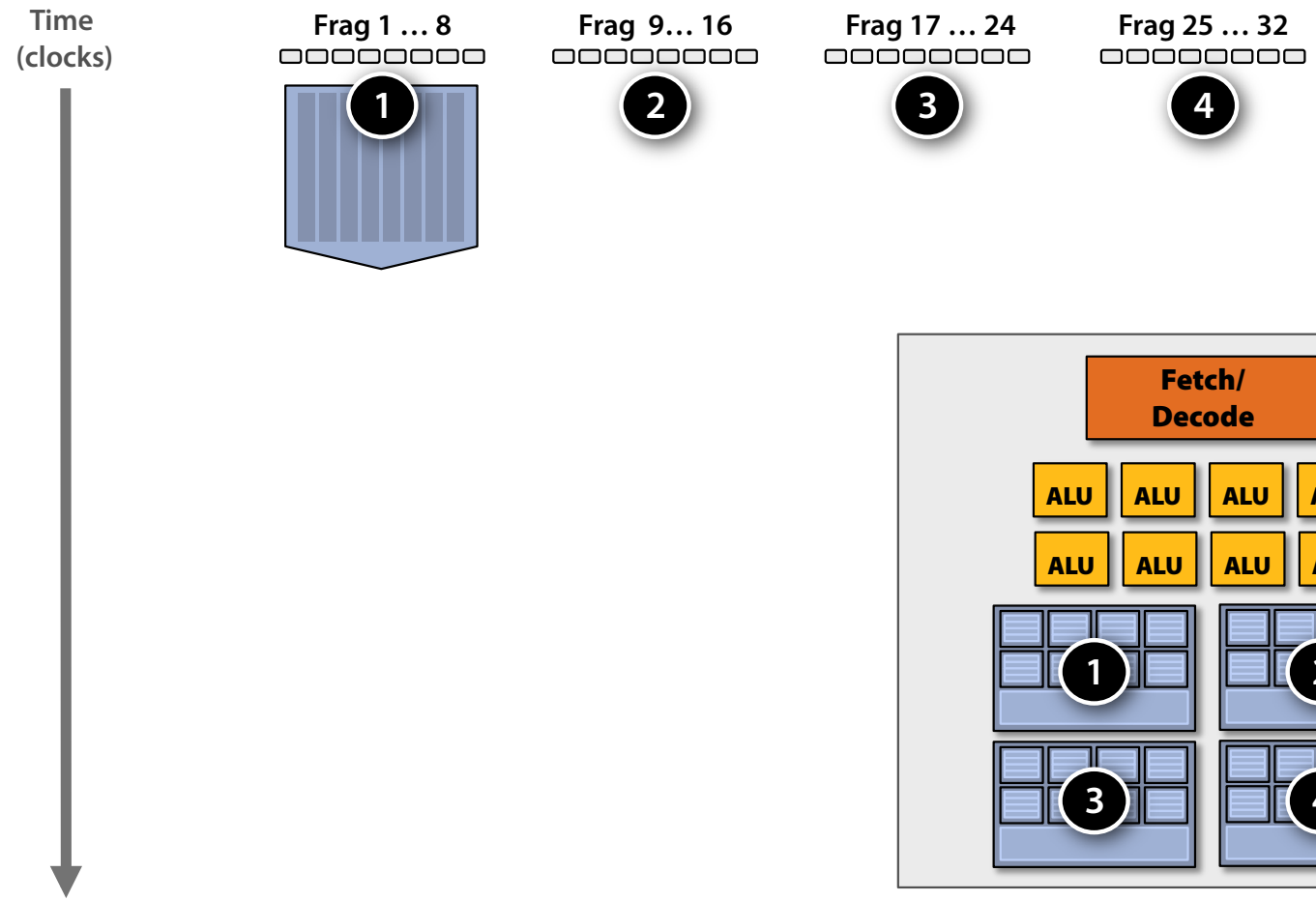


Idea #3

	Even more parallelism
+	Some extra memory
=	A solution!

credit: Andreas Klöckner (NYU)

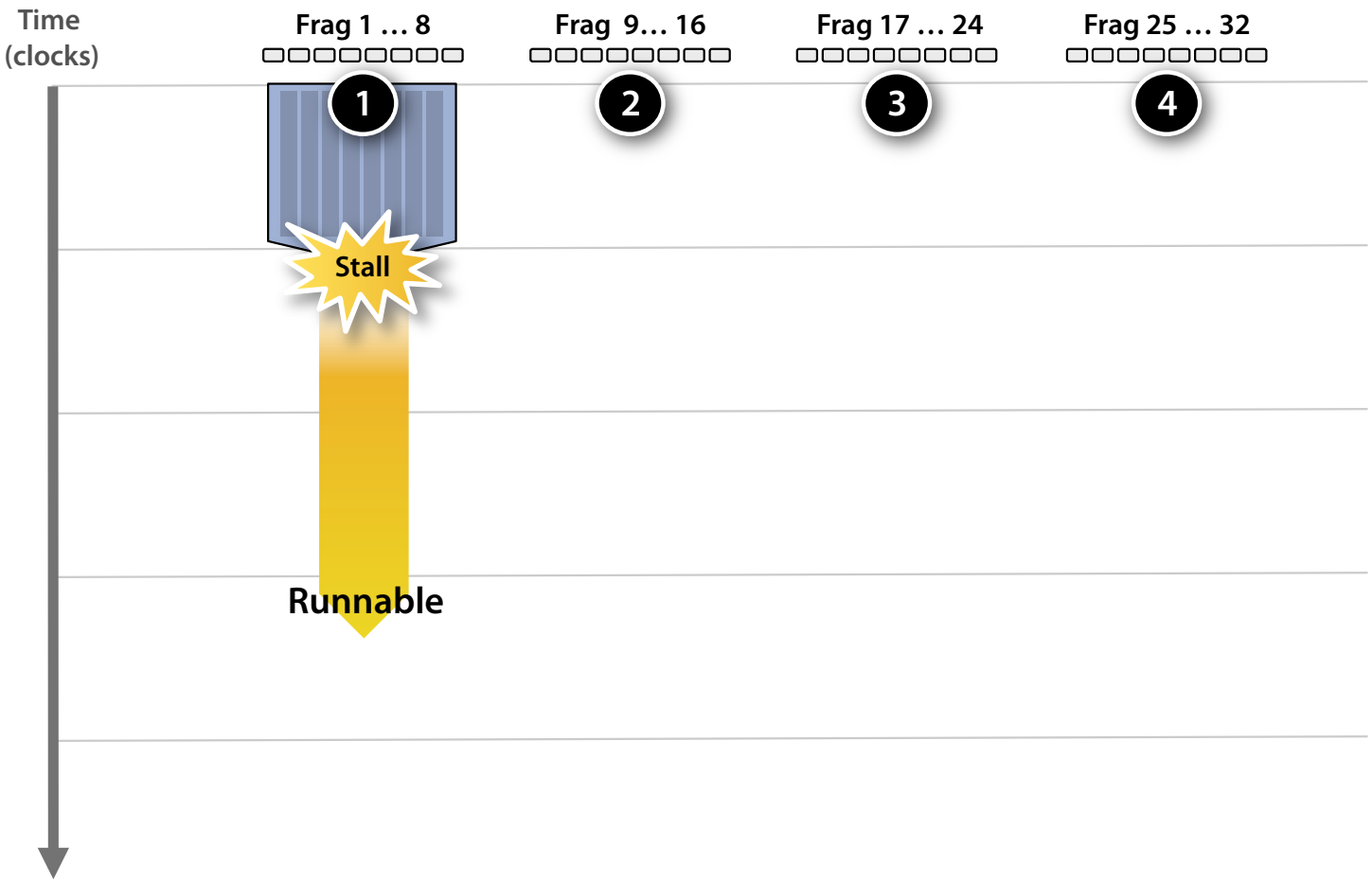
Hiding Memory Latency



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

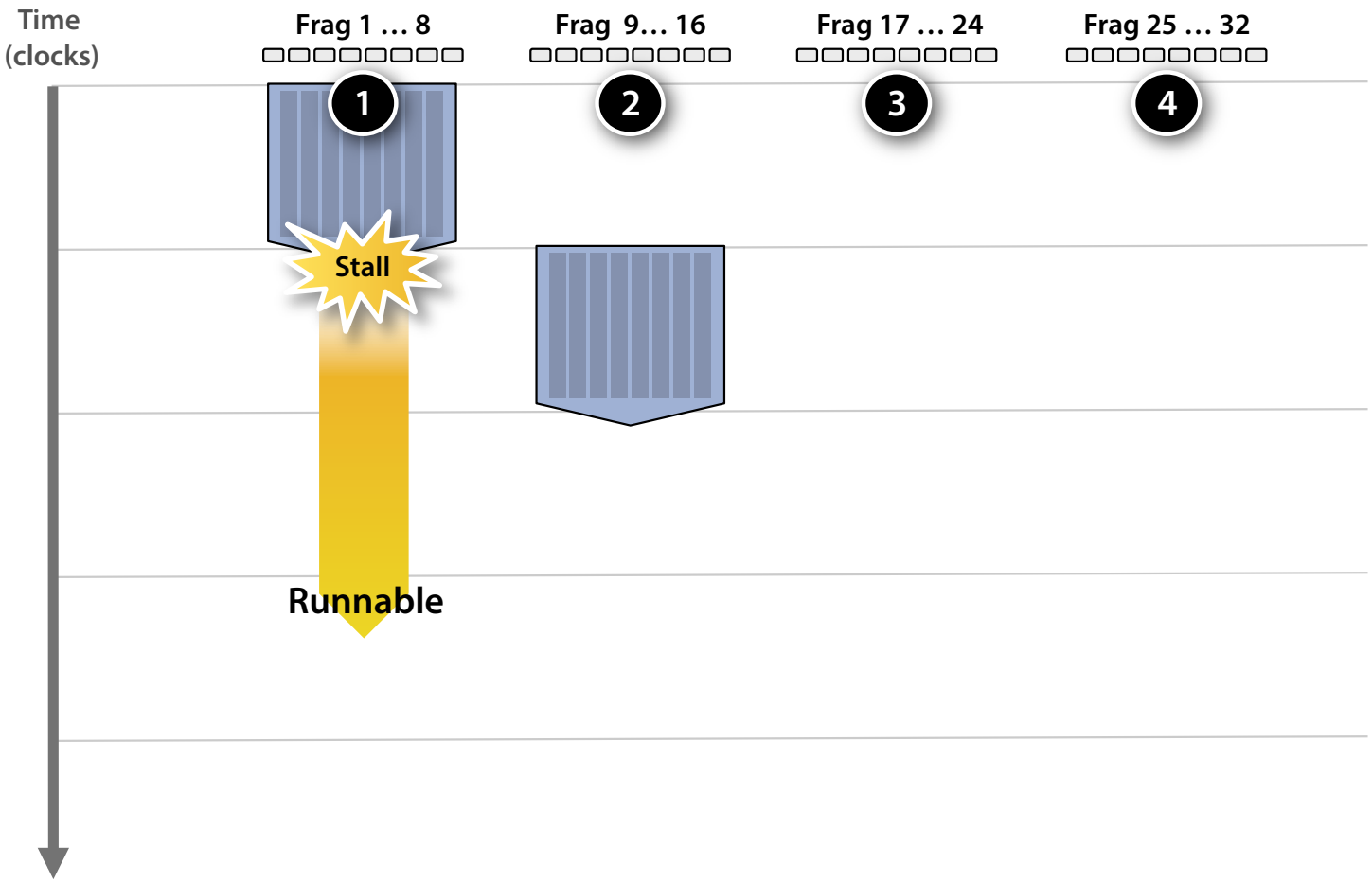
Hiding Memory Latency



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

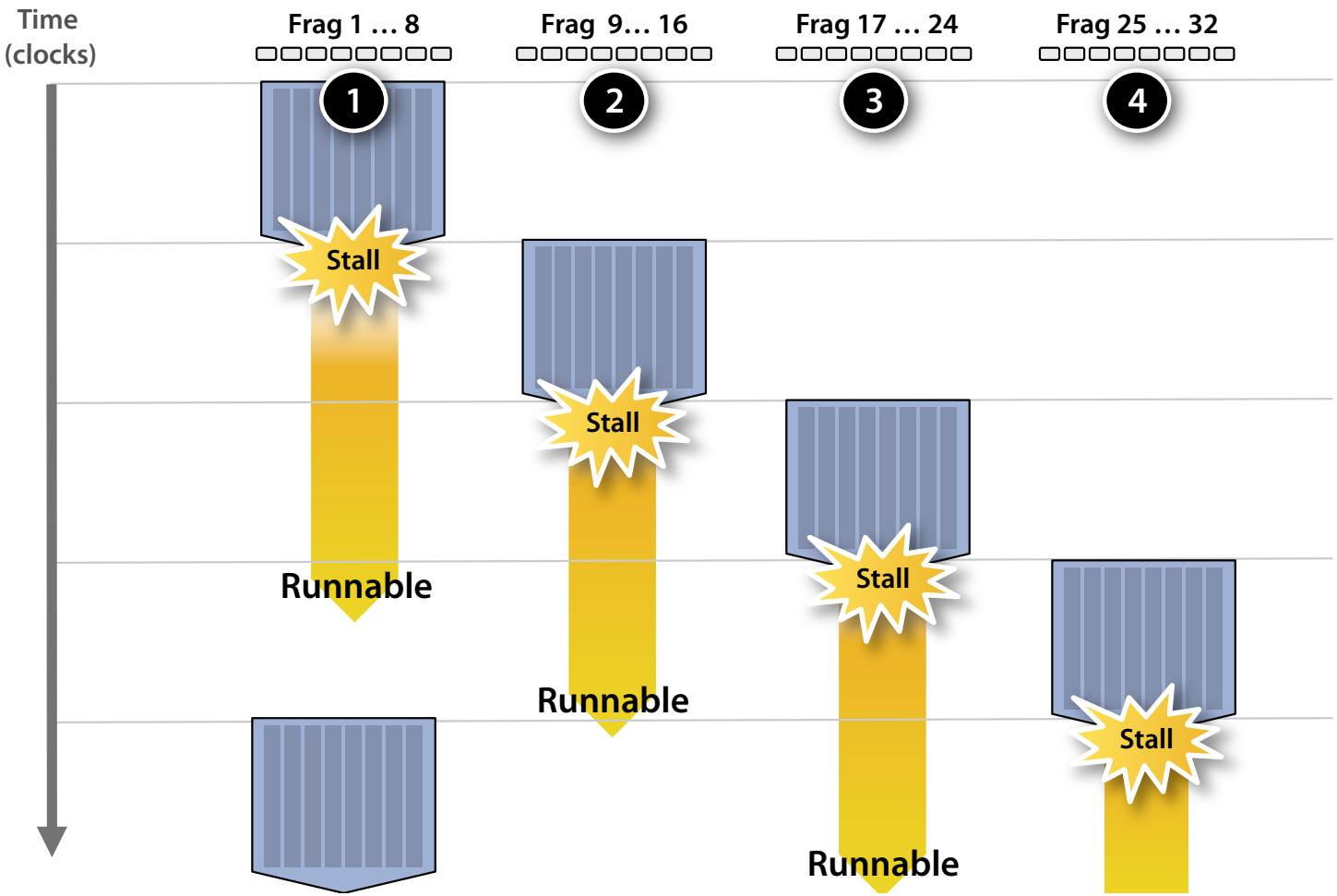
Hiding Memory Latency



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

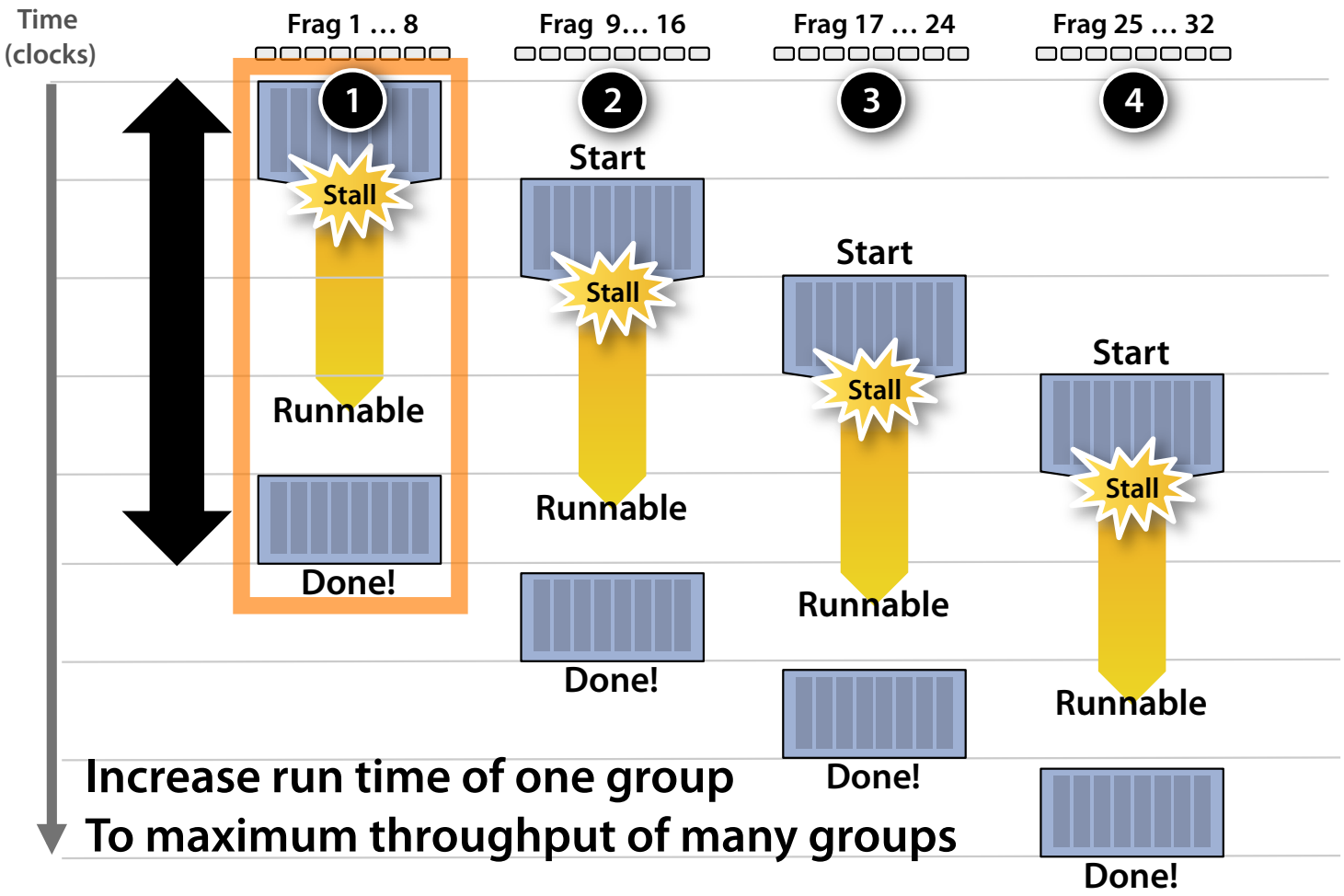
Hiding Memory Latency



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

Hiding Memory Latency



Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

GPU Architecture Summary

Core Ideas:

1. Many slimmed down cores
   $\rightarrow$ lots of parallelism

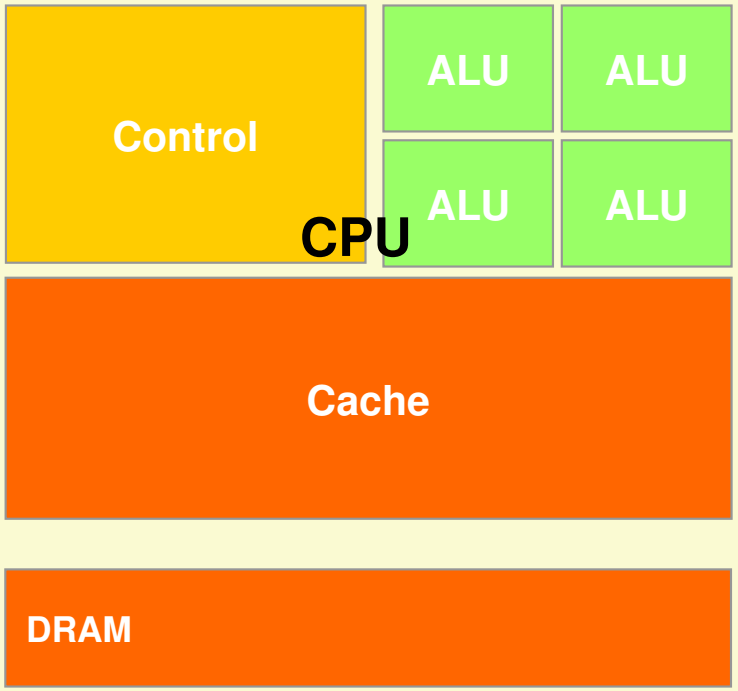
2. More ALUs, Fewer Control Units

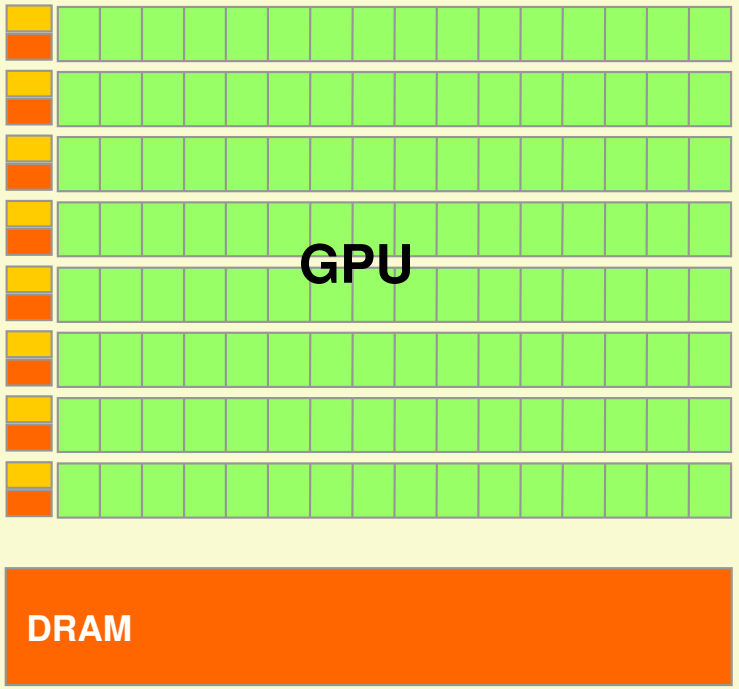
3. Avoid memory stalls by interleaving execution of SIMD groups

Credit: Kayvon Fatahalian (Stanford)

credit: Andreas Klöckner (NYU)

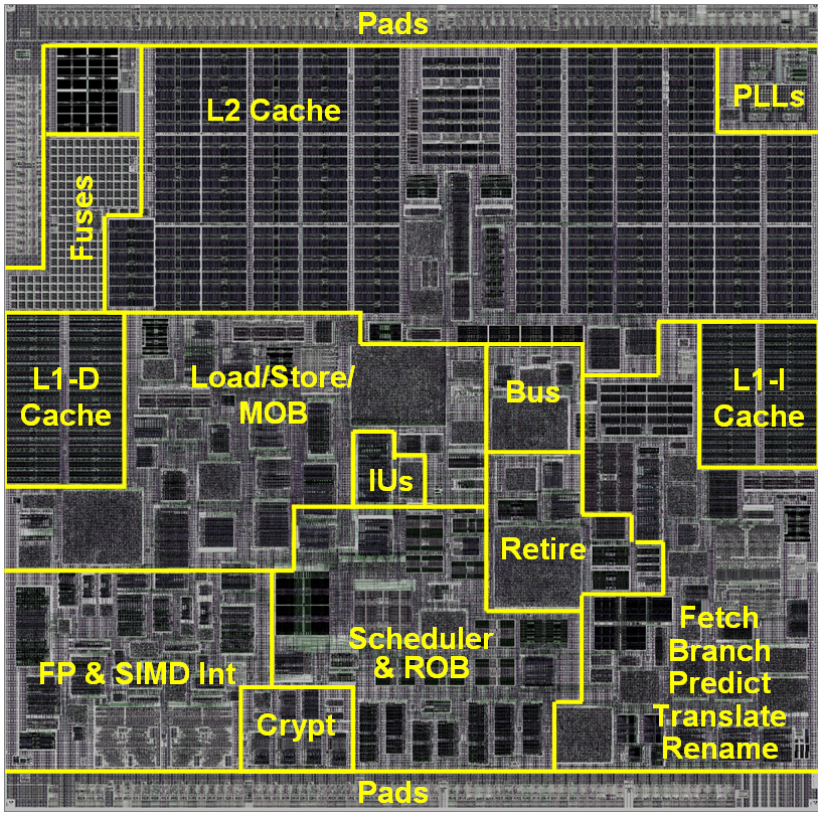
CPUs vs GPUs



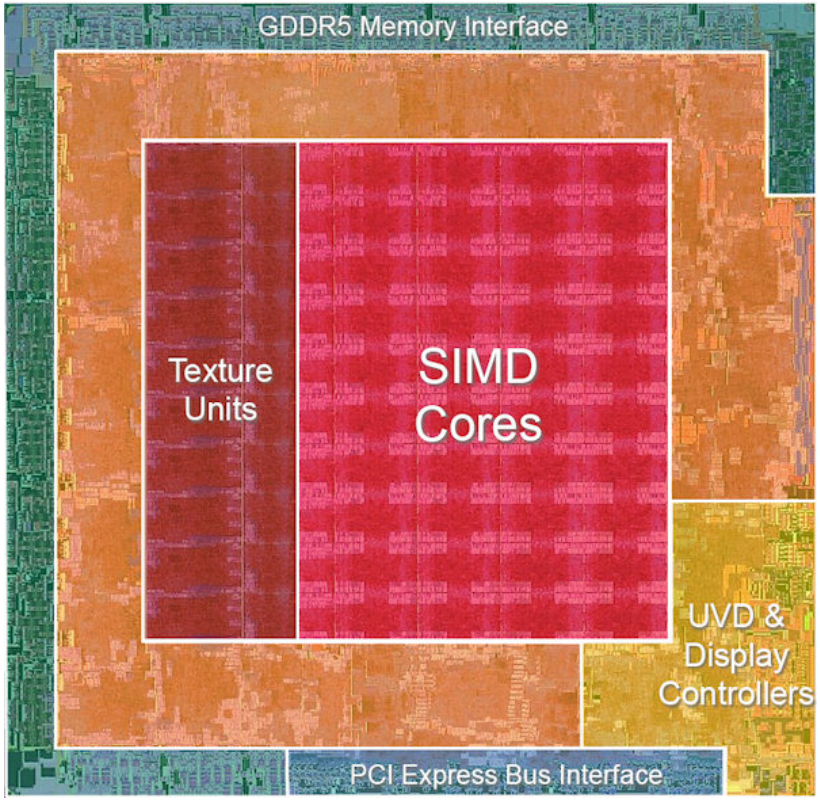


credit: Kirk and Hwu (UIUC)

GPU-CPU Bird's Eye Comparison



Floorplan: VIA Isaiah (2008)
65 nm, 4 SP ops at a time, 1 MiB L2.



Floorplan: AMD RV770 (2008)
55 nm, 800 SP ops
at a time.

credit: Andreas Klöckner (NYU)

GPU Programming: Gains and Losses

Gains	Losses
⊕ Memory Bandwidth (140 GB/s vs. 12 GB/s)	⊖ No performance portability
⊕ Compute Bandwidth (Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)	⊖ Data size $\rightleftarrows$ Alg. design
⊙ Data-parallel programming	⊖ Cheap branches (i.e. `ifs`)
⊙ Functional portability between devices (via OpenCL)	⊖ Fine-grained `malloc`/`free`*)
	⊖ Recursion *)
	⊖ Function pointers *)
	⊖ IEEE 754 FP compliance *)

*) Less problematic with new hardware. (Nvidia "Fermi")

credit: Andreas Klöckner (NYU)

What is OpenCL?

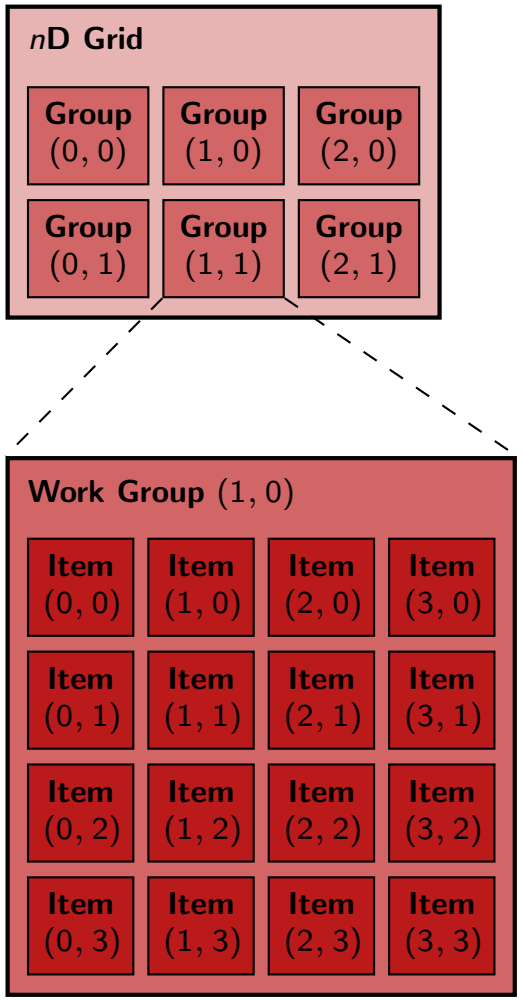
OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]

- Device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU)
- Vendor-neutral
- Comes with RTCG

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)

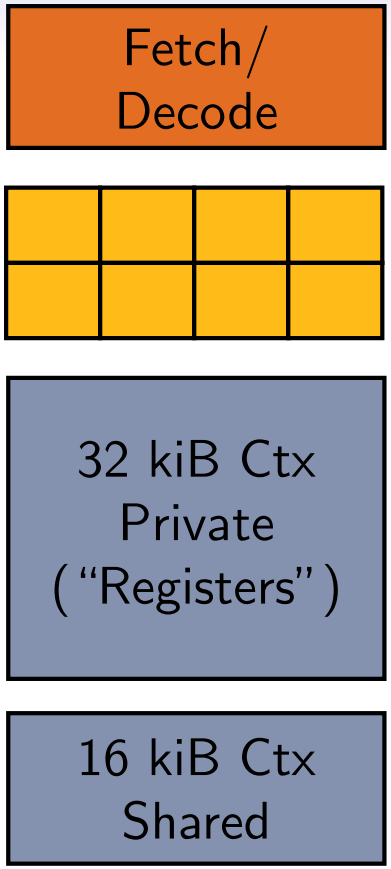
OpenCL: Execution Model



- Two-tiered Parallelism
  - Grid $= N_x \times N_y \times N_z$ work groups
  - Work group $= S_x \times S_y \times S_z$ work items
  - Total: $\prod_{i \in \{x,y,z\}} S_i N_i$ work items
- Abstraction of core/SIMD lane HW concept
- Comm/Sync only within work group
- Grid/Group $\approx$ outer loops in an algorithm
- Device Language: `get_{global,group,local}_{id,size}(axis)`

credit: Andreas Klöckner (NYU)

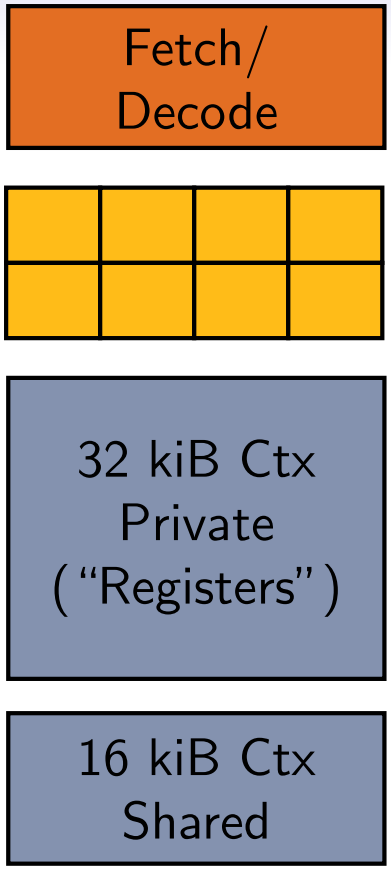
Connection: Hardware $\leftrightarrow$ Programming Model

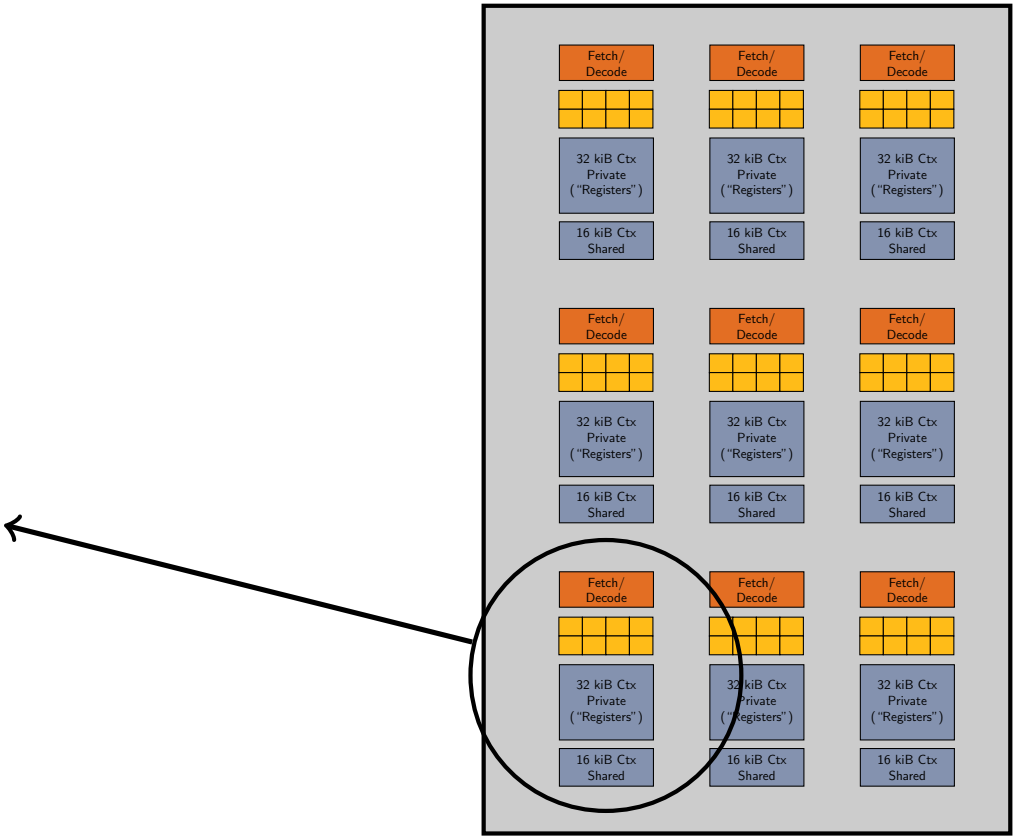


GPU Ideas:

1. Slim down each core
   $\rightarrow$ more of them
2. Less Decode, more ALU
3. Waiting for memory?
   Just work on something else.
   $\rightarrow$ extra context storage

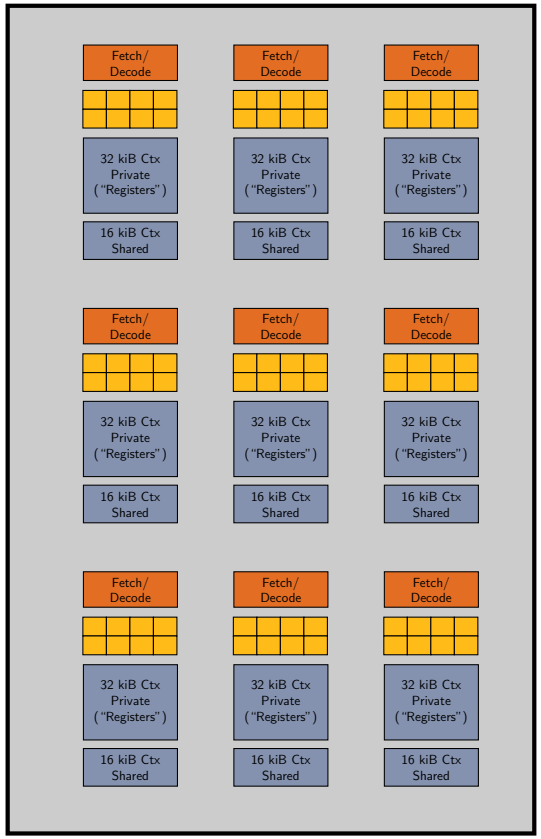
Connection: Hardware $\leftrightarrow$ Programming Model





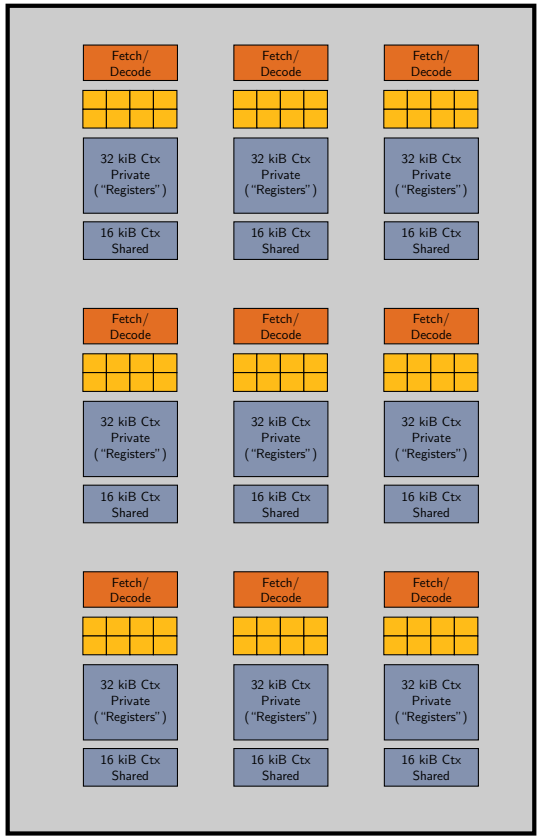
credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model



credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model



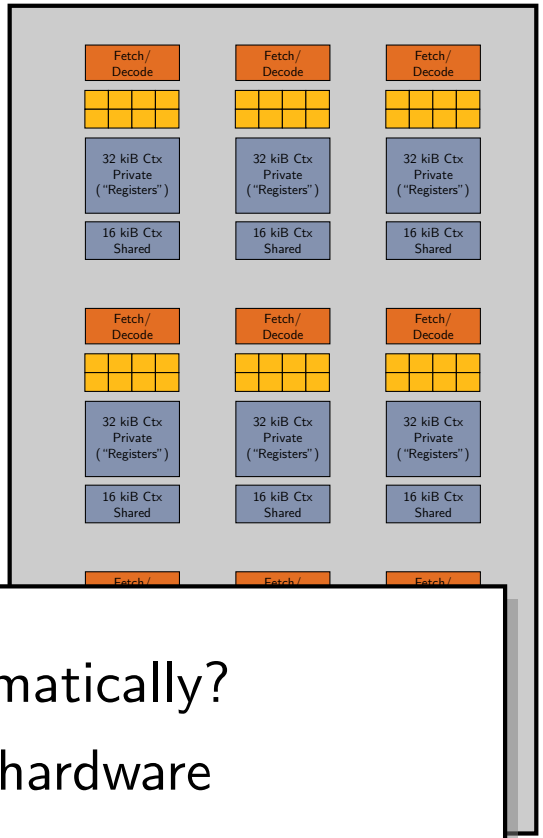
Idea:

- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model

Who cares how many cores?



Idea

-
-

Consider: Which is easy to do automatically?

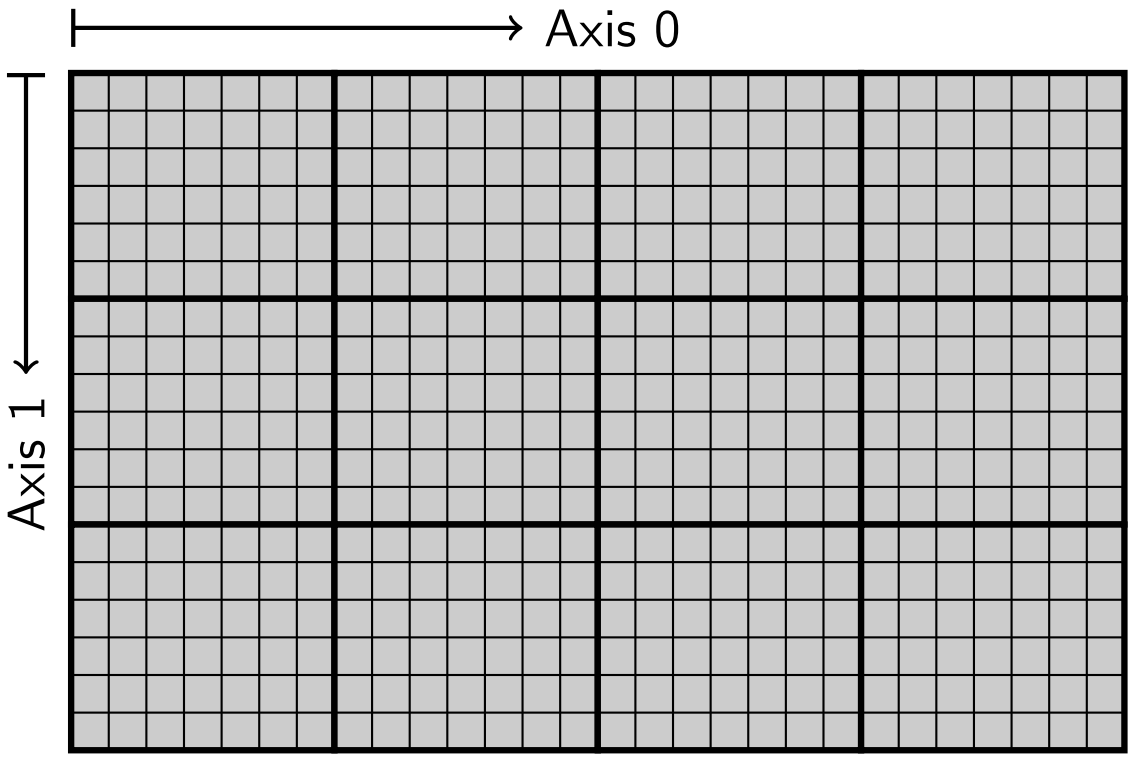
- Parallel program $\rightarrow$ sequential hardware

or

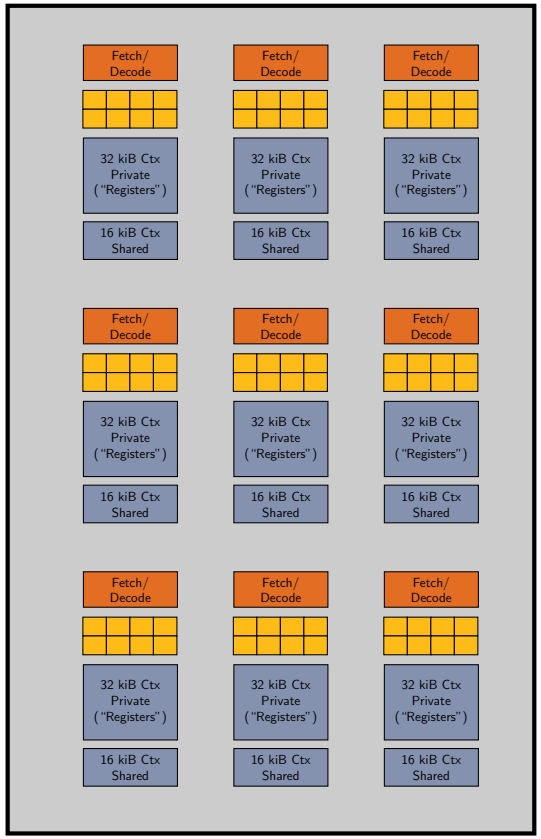
- Sequential program $\rightarrow$ parallel hardware?

credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model



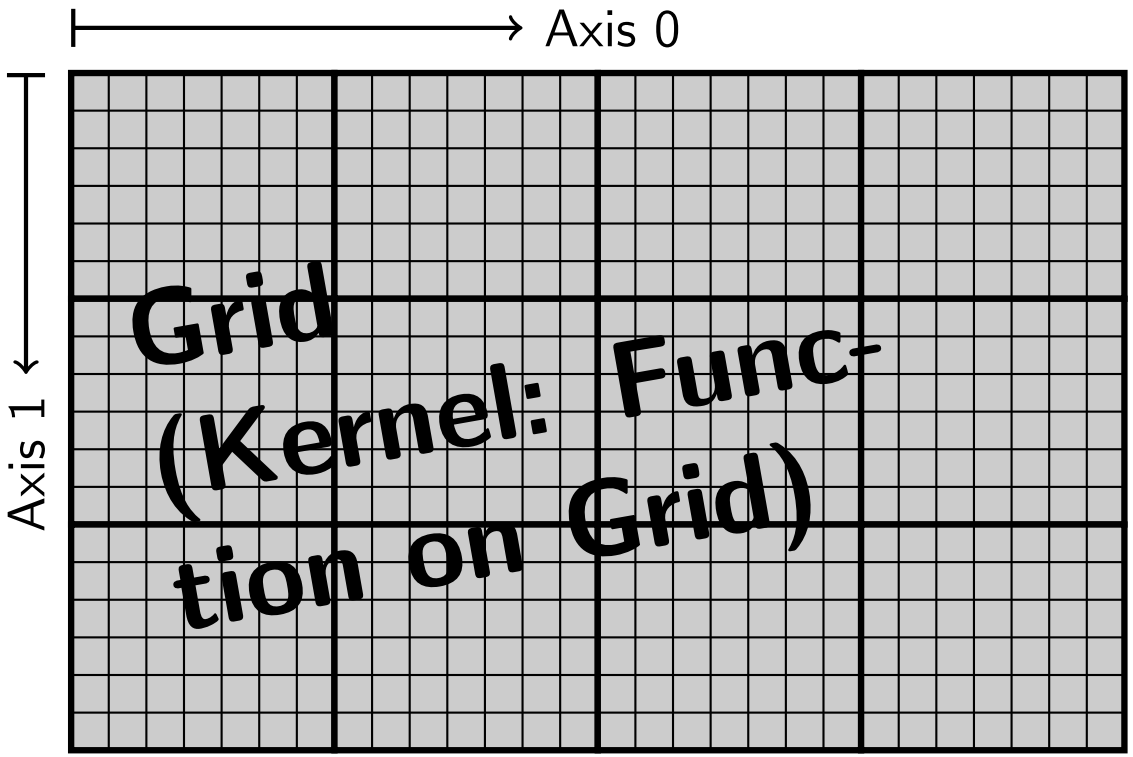
Software representation



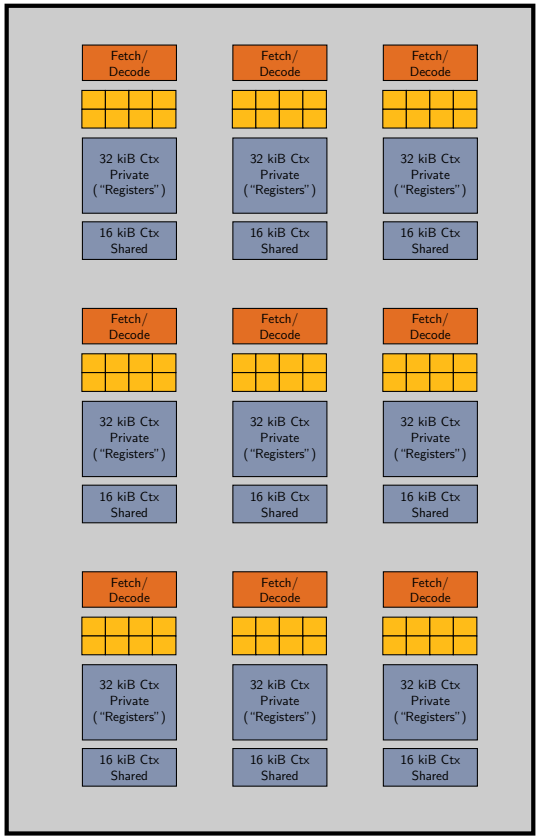
Hardware

credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model

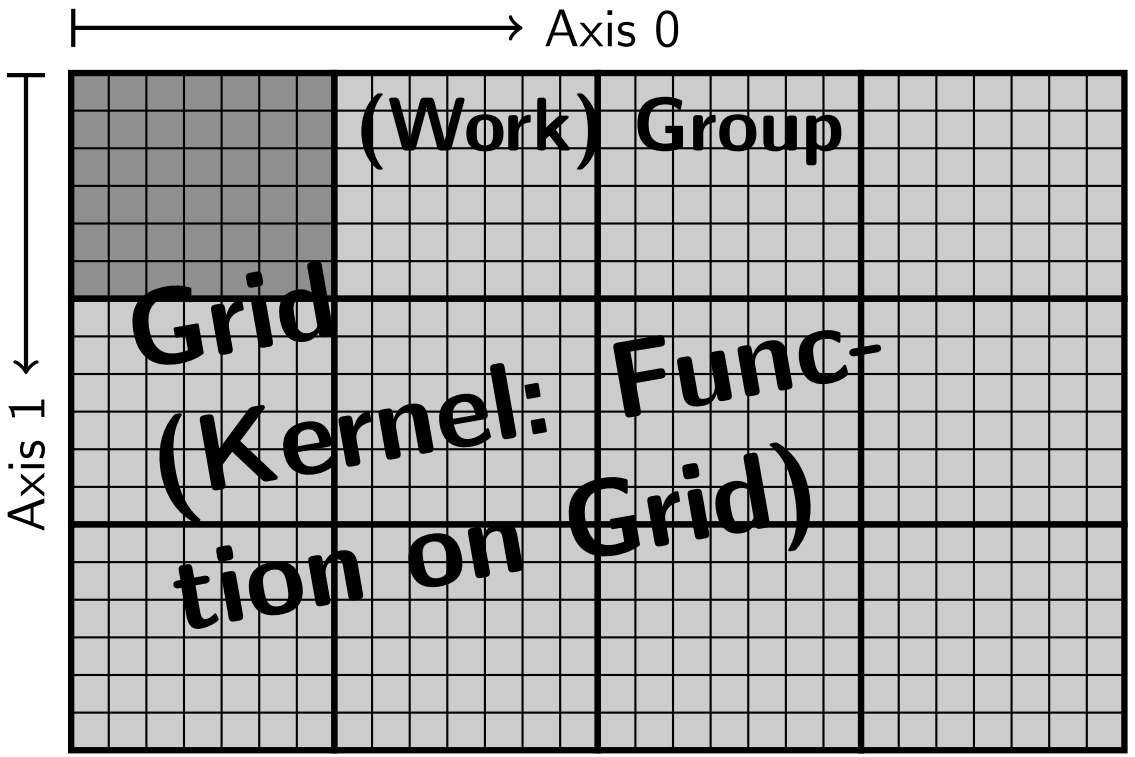


Software representation

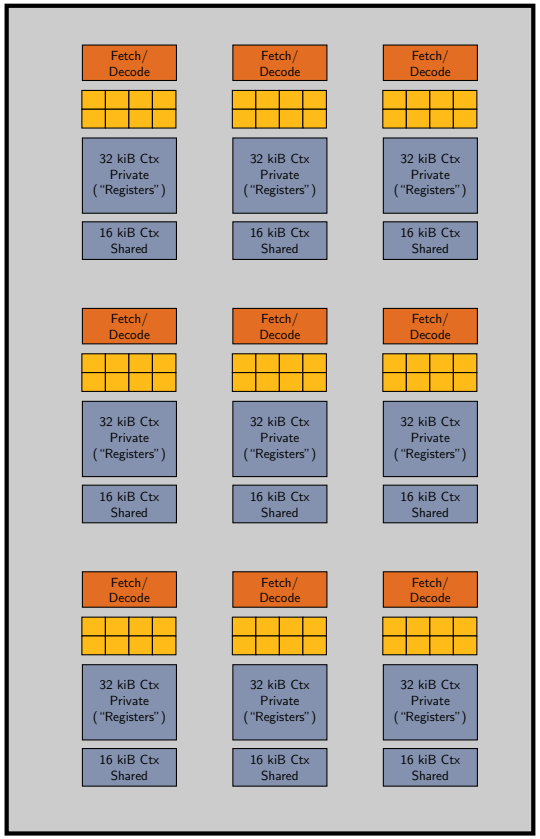


Hardware

Connection: Hardware $\leftrightarrow$ Programming Model



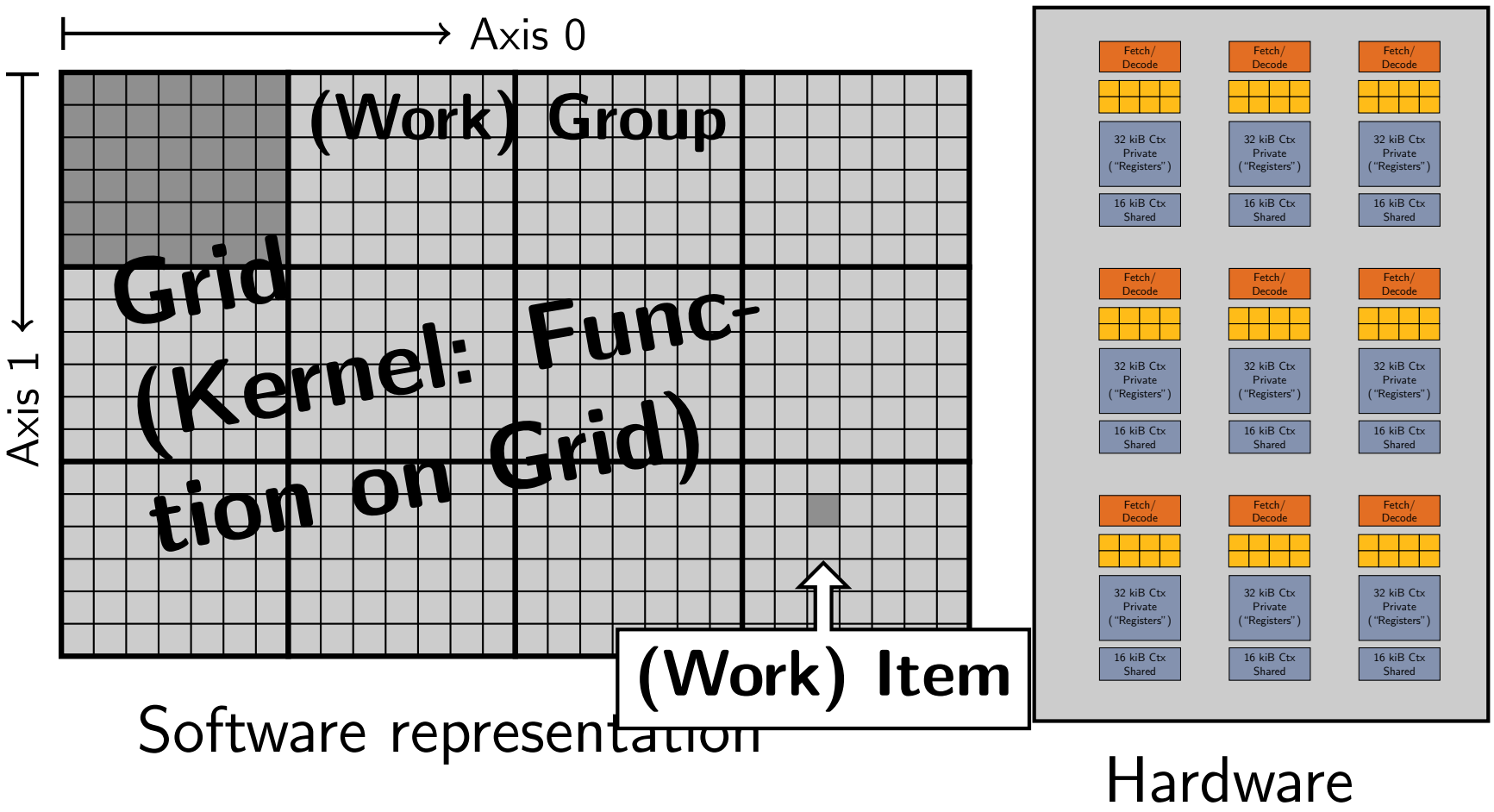
Software representation



Hardware

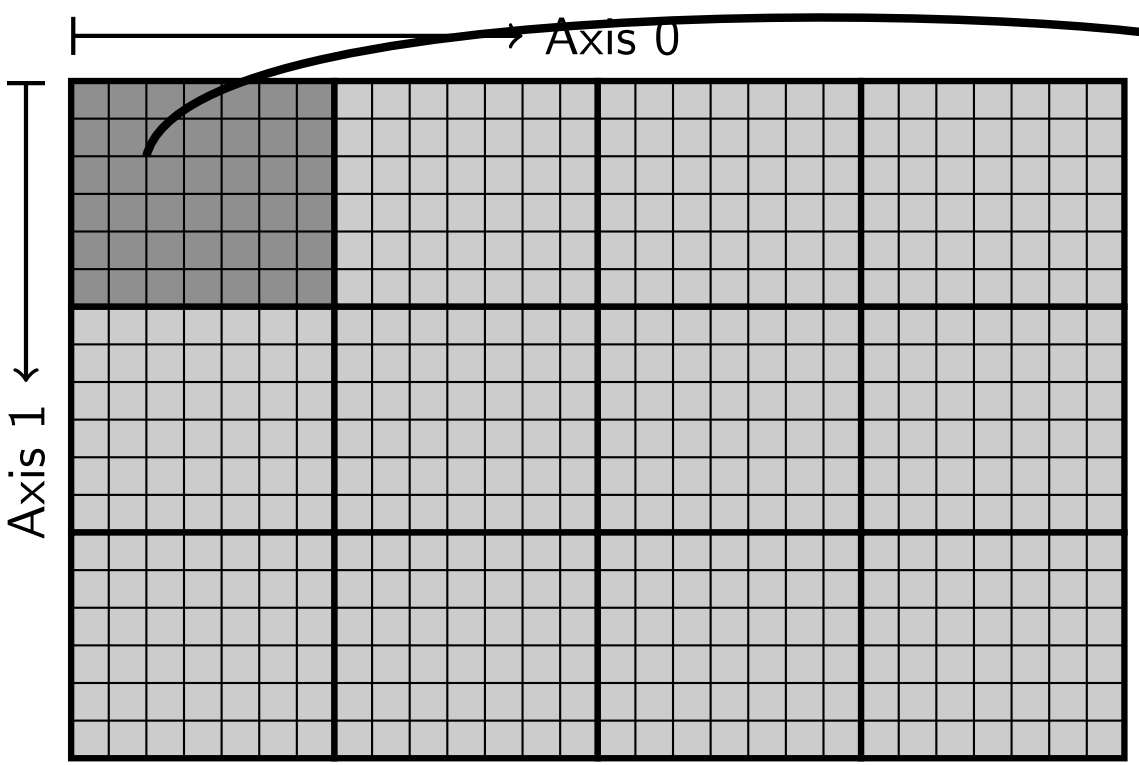
credit: Andreas Klöckner (NYU)

Connection: Hardware ↔ Programming Model

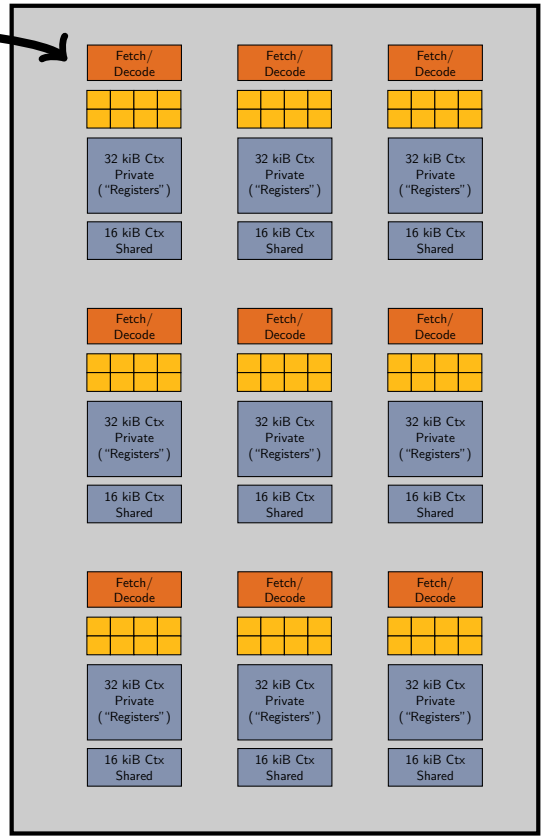


credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model



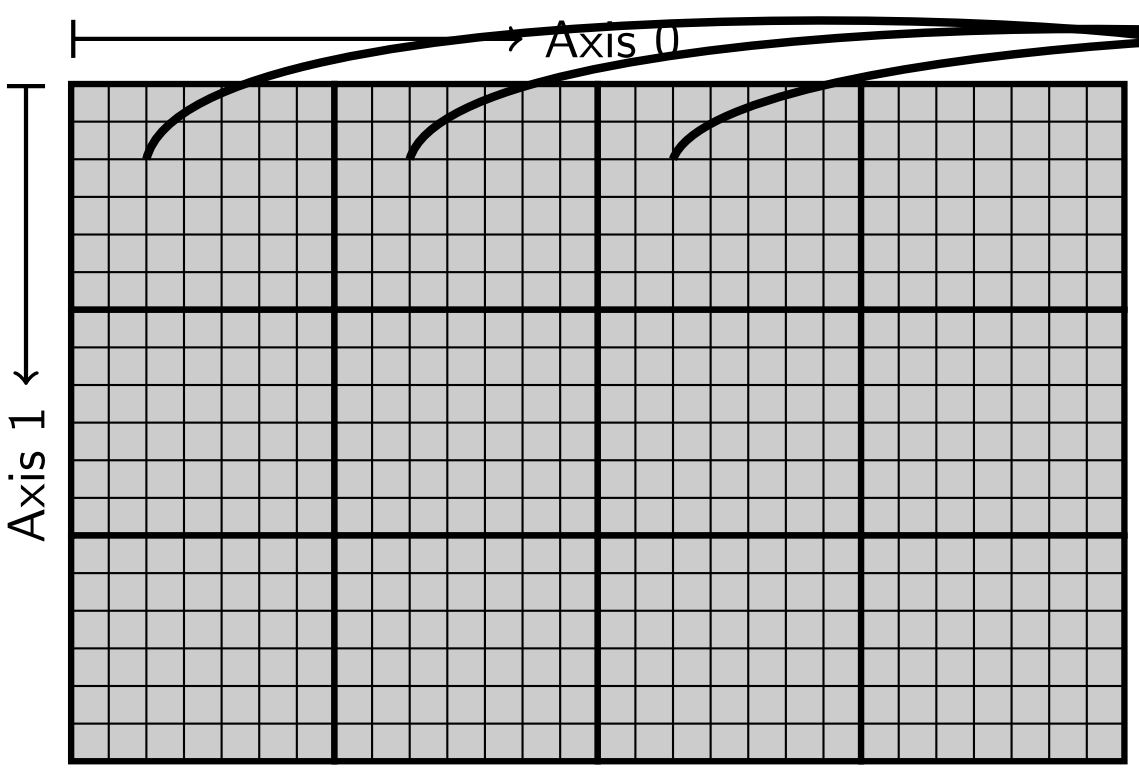
Software representation



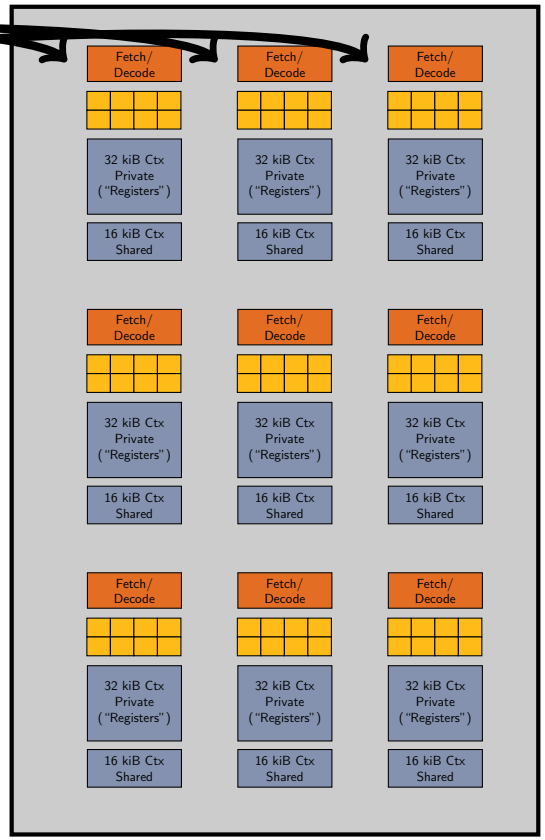
Hardware

credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model



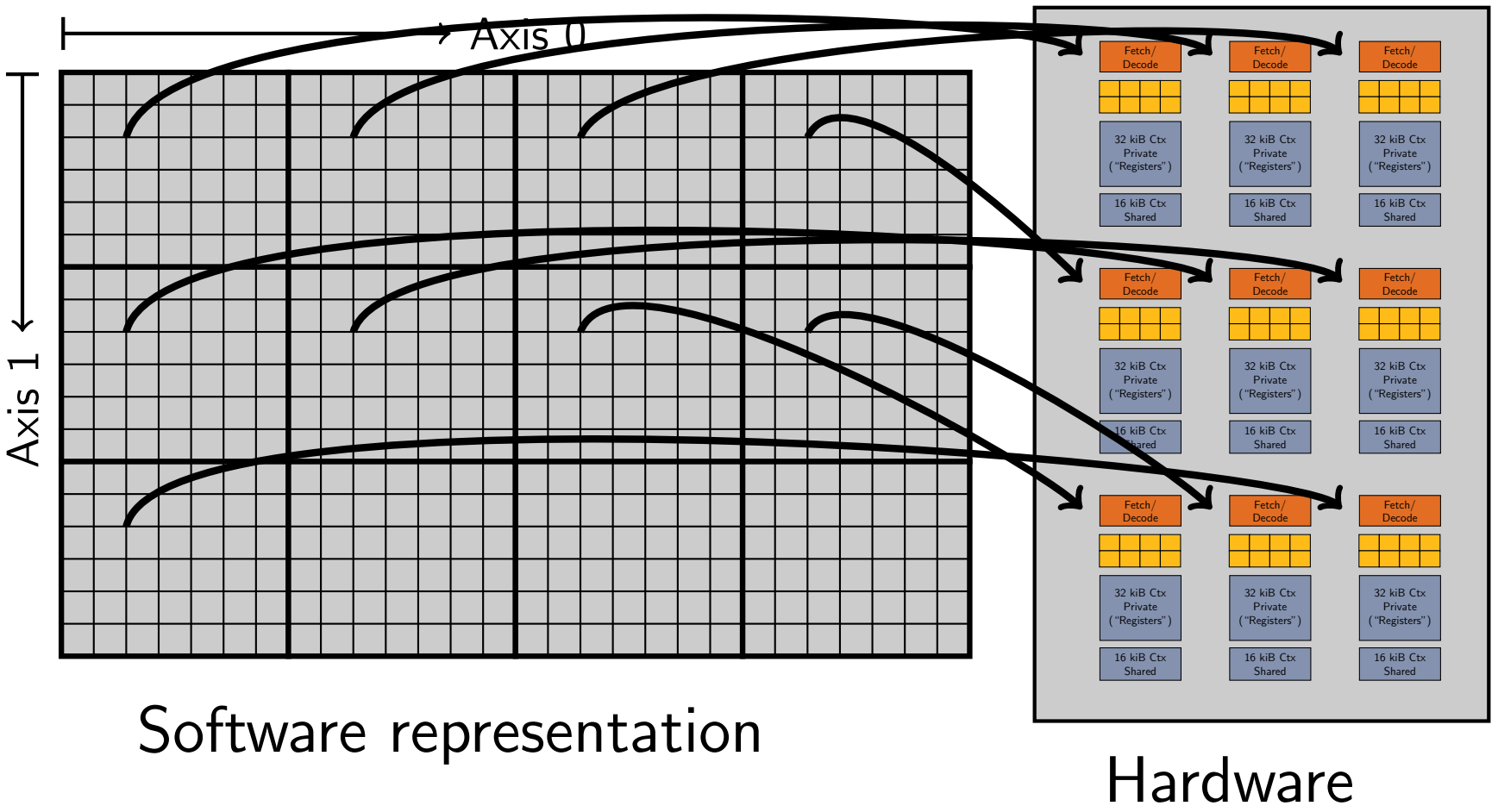
Software representation



Hardware

credit: Andreas Klöckner (NYU)

Connection: Hardware $\leftrightarrow$ Programming Model



credit: Andreas Klöckner (NYU)

Dive into OpenCL: Preparation

```
1  #ifdef __APPLE__
2  #include <OpenCL/opencl.h>
3  #else
4  #include <CL/cl.h>
5  #endif
```

```
1  #include "cl-helper.h"
2
3  int main()
4  {
5    // init
6    cl_context ctx; cl_command_queue queue;
7    create_context_on("NVIDIA", NULL, 0, &ctx, &queue, 0);
8
9    // allocate and initialize CPU memory
10   const size_t sz = 10000;
11   float a[sz];
12   for (size_t i = 0; i < sz; ++i) a[i] = i;
```

credit: Andreas Klöckner (NYU)

Dive into OpenCL: Memory

```
15  // allocate GPU memory, transfer to GPU
16
17  cl_int status;
18  cl_mem buf_a = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
19      sizeof(float) * sz, 0, &status);
20  CHECK_CL_ERROR(status, "clCreateBuffer");
21
22  CALL_CL_GUARDED(clEnqueueWriteBuffer, (
23      queue, buf_a, /*blocking*/ CL_TRUE, /*offset*/ 0,
24      sz * sizeof(float), a,
25      0, NULL, NULL));
```

credit: Andreas Klöckner (NYU)

Dive into OpenCL: Running

```
28  // load kernels
29  char *knl_text = read_file("twice.cl");
30  cl_kernel knl = kernel_from_string(ctx, knl_text, "twice", NULL);
31  free(knl_text);
32
33  // run code on GPU
34  SET_1_KERNEL_ARG(knl, buf_a);
35  size_t gdim[] = { sz };
36  size_t ldim[] = { 1 };
37  CALL_CL_GUARDED(clEnqueueNDRangeKernel,
38      (queue, knl,
39       /*dimensions*/ 1, NULL, gdim, ldim,
40       0, NULL, NULL));
```

```
1  __kernel void twice(__global float *a)
2  { a[get_global_id(0)] *= 2; }
```

credit: Andreas Klöckner (NYU)

Dive into OpenCL: Clean-up

```
43    // clean up ...
44    CALL_CL_GUARDED(clReleaseMemObject, (buf_a));
45    CALL_CL_GUARDED(clReleaseKernel, (knl));
46    CALL_CL_GUARDED(clReleaseCommandQueue, (queue));
47    CALL_CL_GUARDED(clReleaseContext, (ctx));
48  }
```

Why check for errors?

- GPUs have (some) memory protection
- Invalid sizes (block/grid/...)
- Out of memory, access restriction, hardware limitations, etc.

credit: Andreas Klöckner (NYU)

All this boiler-plate code looks tedious!

Photo: David Warde-Farley

I ain't afraid of no code!

Andreas Klöckner! (author lead developer: PyCUDA and PyOpenCL)

On this laptop... MeasureGpuarraySpeedRandom.py

Size	Time GPU	Time CPU	GPU vs CPU speedup
1024	0.0642932	2.26417274475e-05	0.000352163410604
2048	0.0621113	2.78759365082e-05	0.000448806024647
4096	0.0620930	5.24587516785e-05	0.00084841582438
8192	0.0618909	9.85365142822e-05	0.0015920992357
16384	0.0624442	0.0002004511108 4	0.003210080362 6
32768	0.0621328	0.000385512207031	0.006204639815 62
65536	0.0623410	0.000756432983398	0.012133782426 4
131072	0.0623857	0.00150555249023	0.024132954885 3
262144	0.0628253	0.00307793188477	0.048991876817 5
524288	0.0635969	0.00626709130859	0.098543940867 3
1048576	0.0639297	0.0123591591797	0.193323946764
2097152	0.0656791	0.024891652832	0.37898890802 9
4194304	0.0677178	0.0569678466797	0.84125343789
8388608	0.0752677	0.121399638672	1.6129035941 2
16777216	0.0892691	0.241773164062	2.7083606958 2

On this laptop...

Creating a random vector of varying lengths (cont'd)

```
Size     |Time GPU |Time CPU           |GPU vs CPU speedup
---------+---------+-------------------+------------------
16777216|0.0878536|0.242302675781     |2.75802712942
33554432|0.1143567|0.483408398438     |4.227196092
67108864|0.1683864|0.9761471875       |5.79706443986
```

Note: here, we're comparing a latest generation Intel Core i5-3320M CPU, which can go up to 3.3 GHz , against a modest mobile NVIDIA NVS 5200M graphics card, with 96 CUDA cores and only a 64 bit memory bus.

Also this is just for making random numbers, and we used a library call to do it, so we haven't actually done anything.
Let's now try to compute sin() elementwise...

Again, on this laptop...

Computing sin() of a bunch of numbers:
modified PyCUDA SimpleSpeedTest.py wiki example

```
Speedups:
---------
21.037    --  SourceModule
19.296    --  ElementWise
3.692     --  ElementWise + Python
1.030     --  GPU-Array
```

Again, these numbers are NVS 5200M vs i5-3320M.
In SimpleSpeedTest.py, it is noted that one gets a 100x speedup for using SourceModule code on a GTX 470 GPU vs X5650 Xeon.

Introducing... PyOpenCL

Same flavor, different recipe:

```
import pyopencl as cl, numpy

a = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a_buf = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_buf, a)

prg = cl.Program(ctx, """
    __kernel void twice(__global float *a)
    {
      int gid = get_global_id(0);
      a[gid] *= 2;
    }""").build()

prg.twice(queue, a.shape, None, a_buf).wait()
```

credit: Andreas Klöckner (NYU)

Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\rightarrow$ complement each other
- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough
- Python + CUDA = **PyCUDA**
- Python + OpenCL = **PyOpenCL**

credit: Andreas Klöckner (NYU)

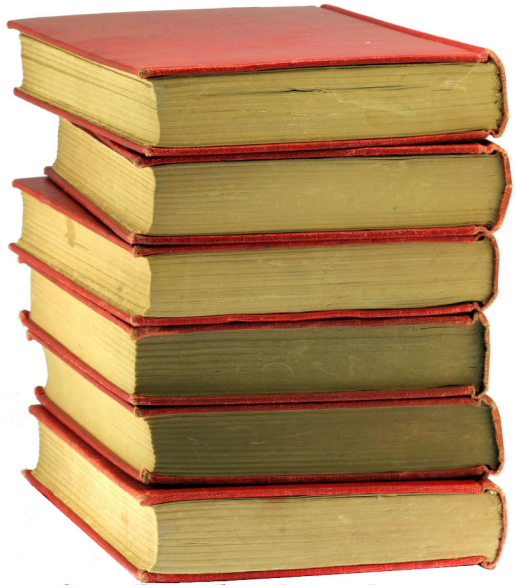
Introducing... PyOpenCL

- PyOpenCL is "PyCUDA for OpenCL"
- Complete, mature API wrapper
- Has: Arrays, elementwise operations, RNG, ...
- Near feature parity with PyCUDA
- Tested on all available Implementations, OSs
- `http://mathema.tician.de/software/pyopencl`

OpenCL

credit: Andreas Klöckner (NYU)

PyOpenCL Philosophy



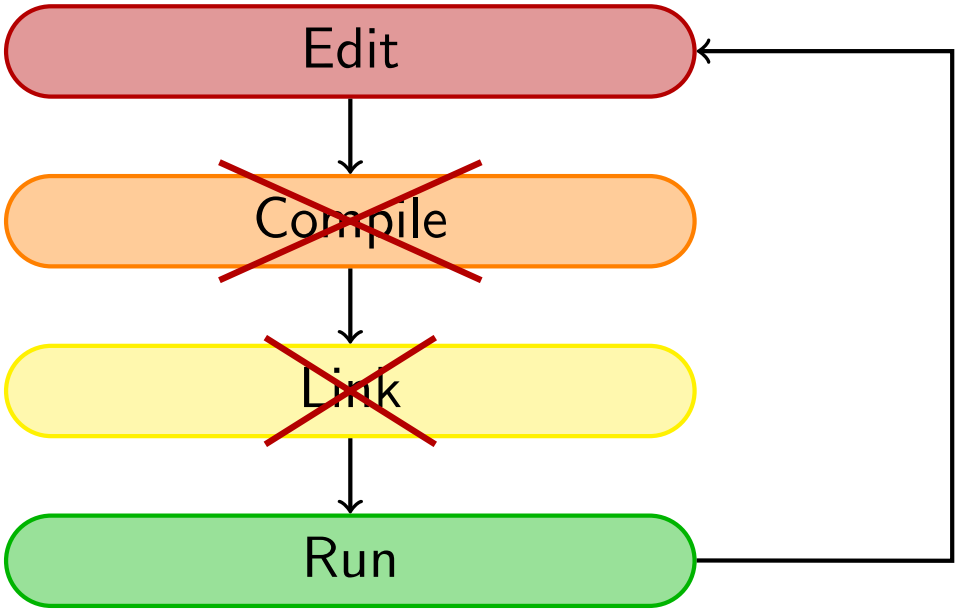
- Provide complete access
- Automatically manage resources
- Provide abstractions
- Allow interactive use
- Check for and report errors automatically
- Integrate tightly with `numpy`

BROWN

credit: Andreas Klöckner (NYU)

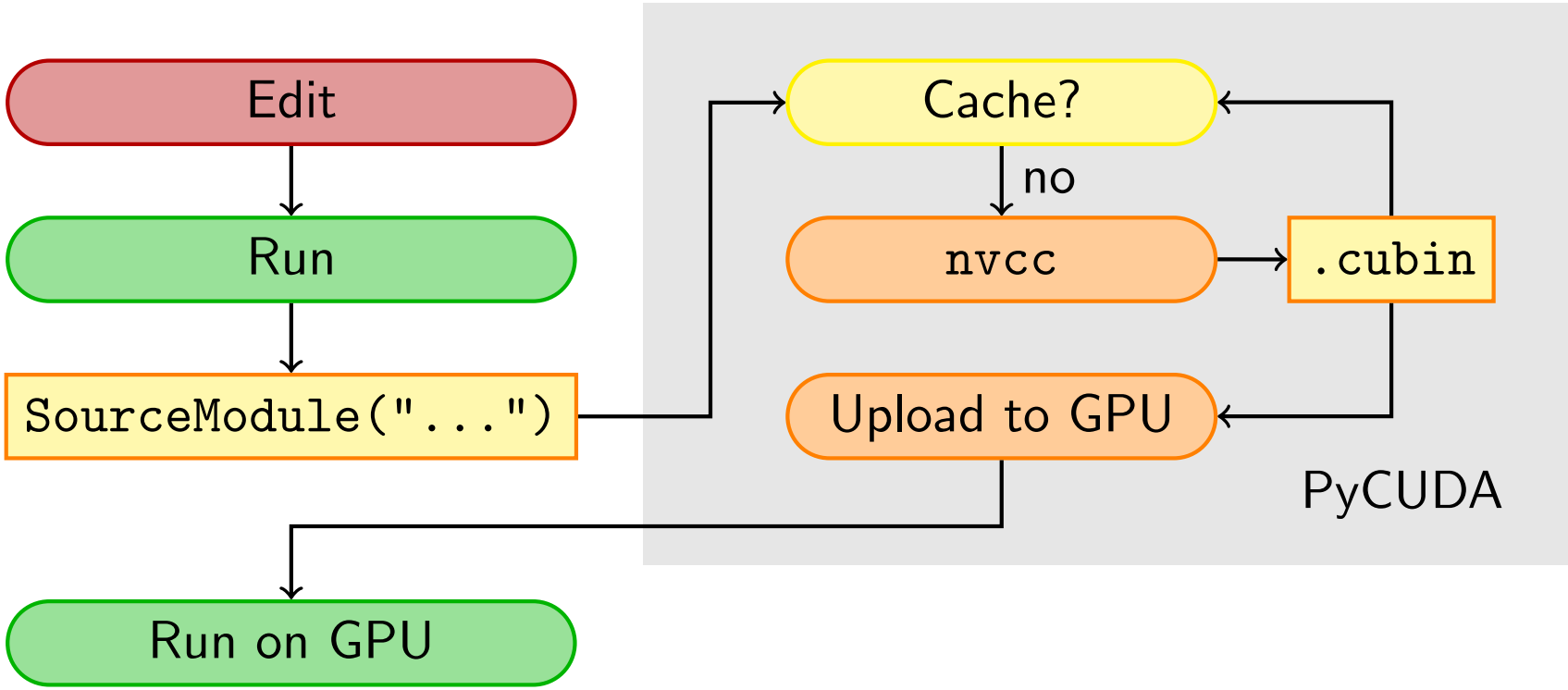
Scripting: Interpreted, not Compiled

Program creation workflow:



credit: Andreas Klöckner (NYU)

PyCUDA: Workflow



credit: Andreas Klöckner (NYU)

GPU Programming: Implementation Choices

- Many difficult questions
- Insufficient heuristics
- Answers are hardware-specific and have no lasting value

credit: Andreas Klöckner (NYU)

GPU Programming: Implementation Choices

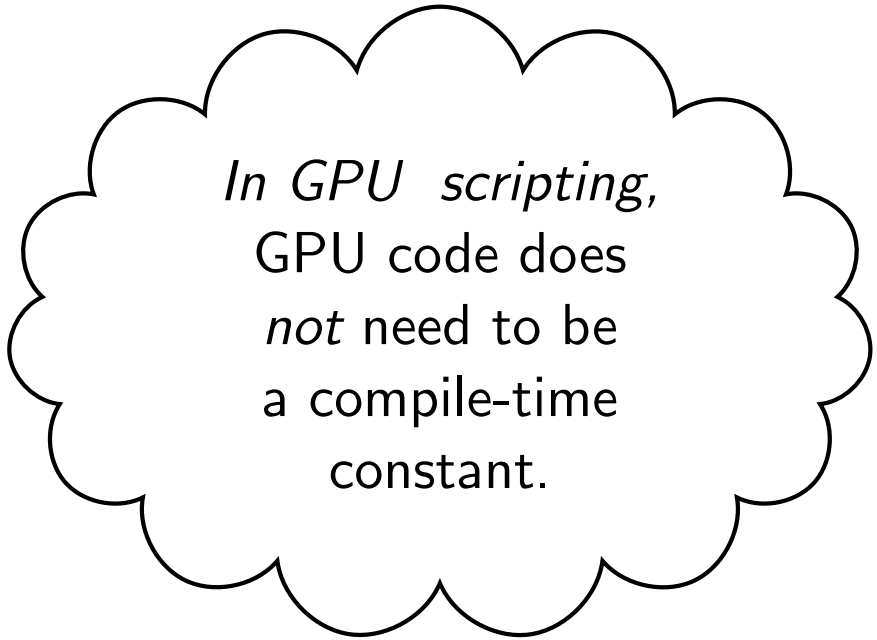
- Many difficult questions
- Insufficient heuristics
- Answers are hardware-specific and have no lasting value

Proposed Solution: Tune automatically for hardware at run time, cache tuning results.

- Decrease reliance on knowledge of hardware internals
- Shift emphasis from tuning *results* to tuning *ideas*

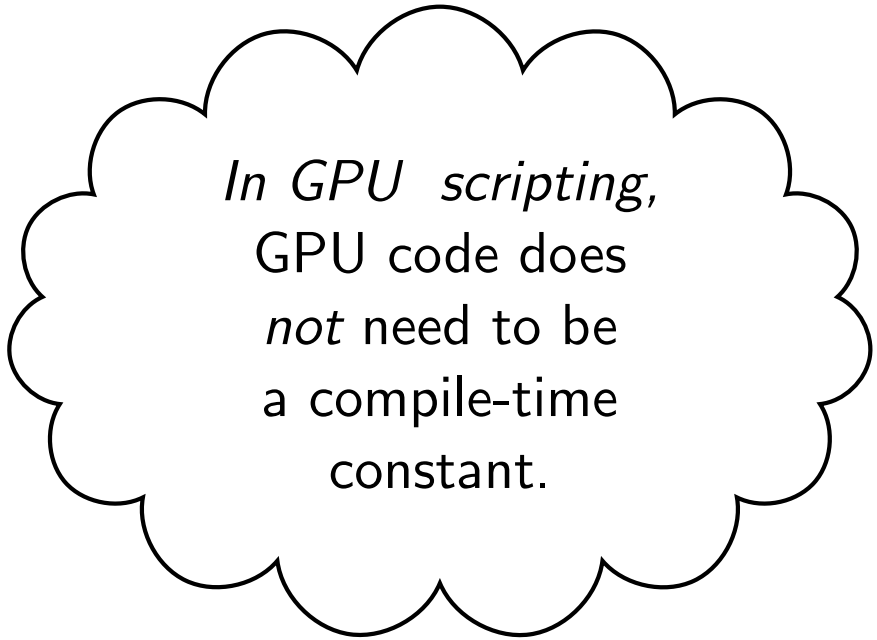
credit: Andreas Klöckner (NYU)

Metaprogramming



credit: Andreas Klöckner (NYU)

Metaprogramming

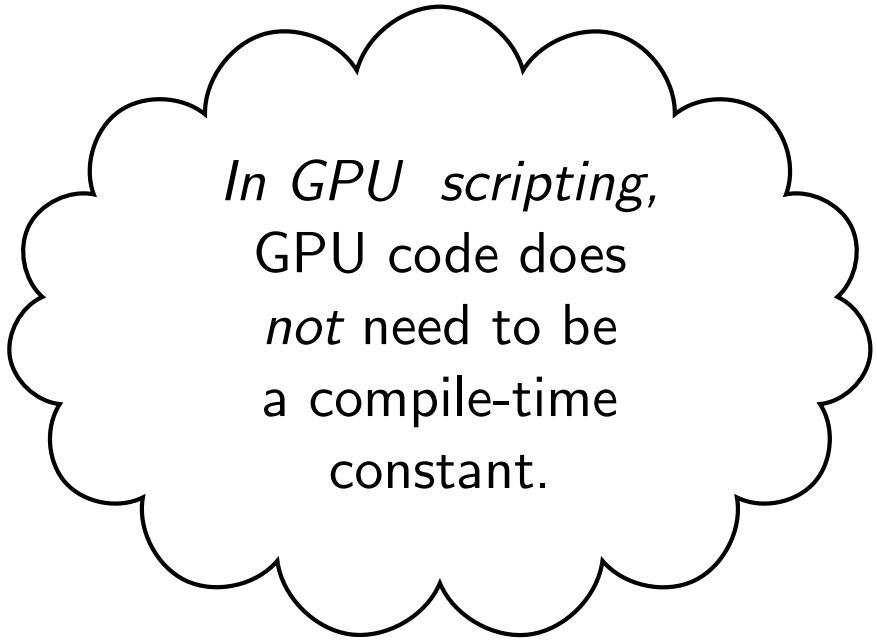


(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Metaprogramming

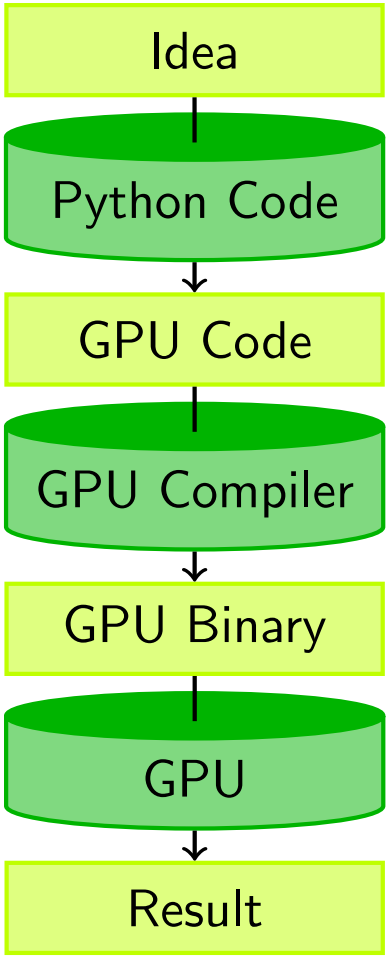
Idea



(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Metaprogramming

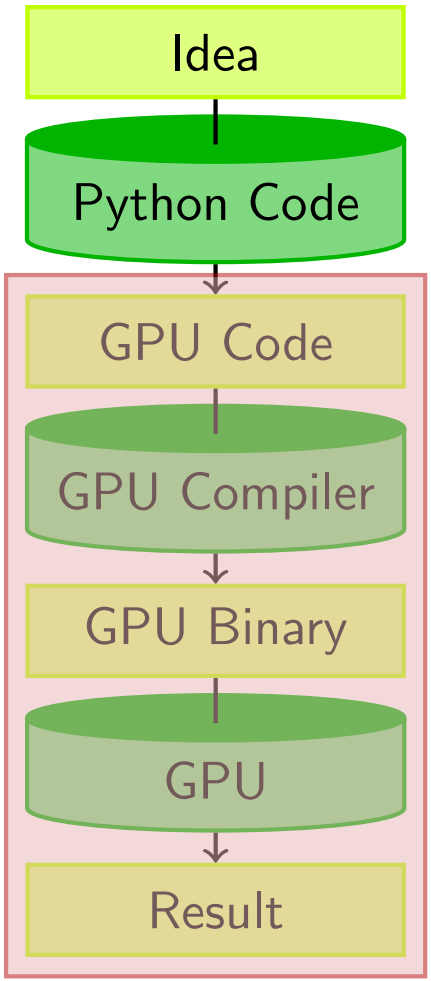


In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Metaprogramming



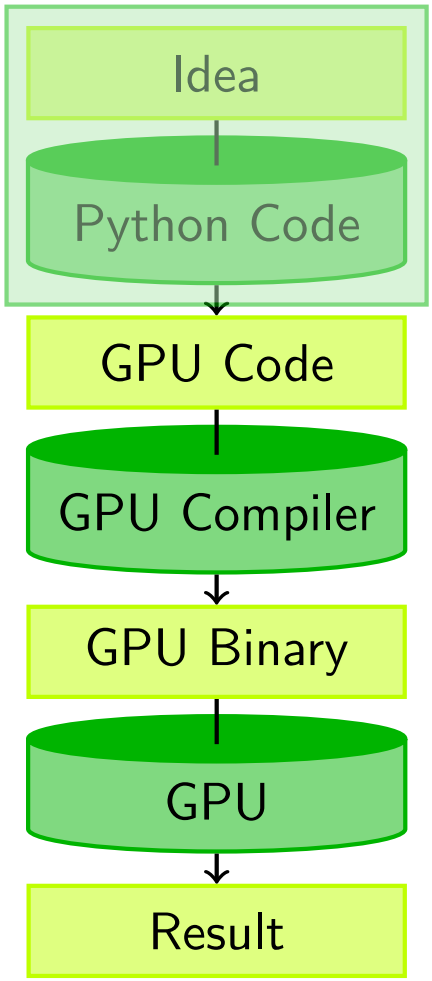
Machine

In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Metaprogramming

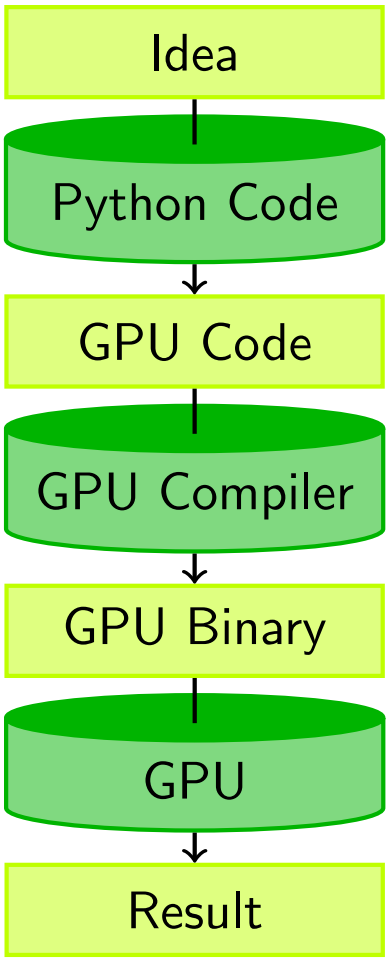


In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Metaprogramming



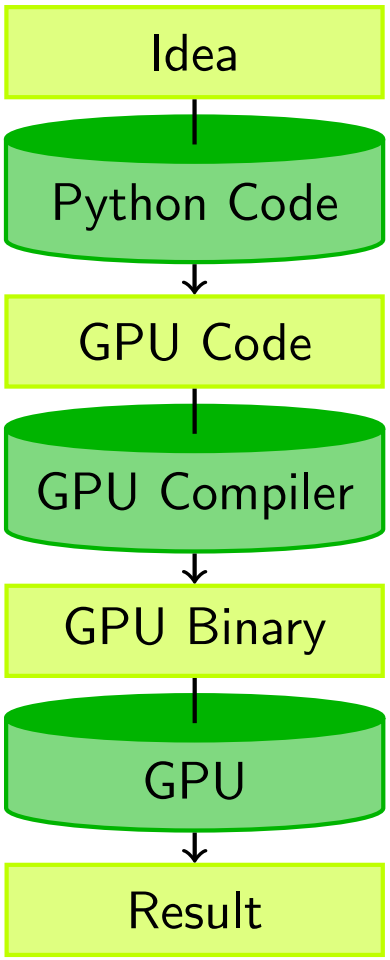
Good for code generation

In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Metaprogramming



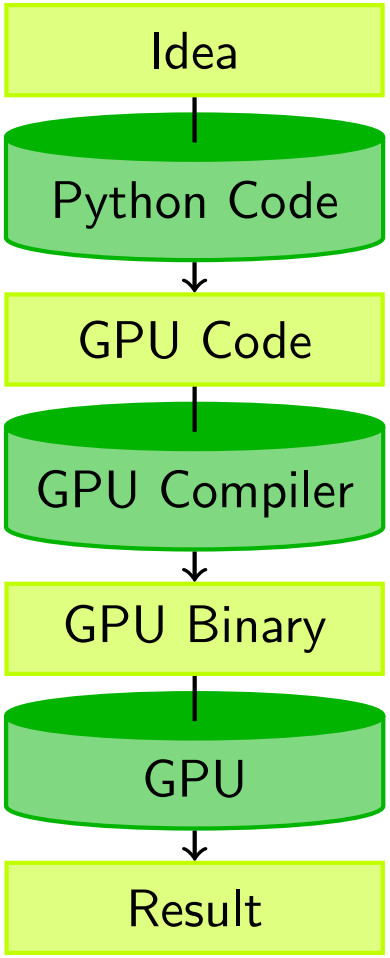
Good for code generation

In PyCUDA GPU code does *not* need to be a compile-time constant.

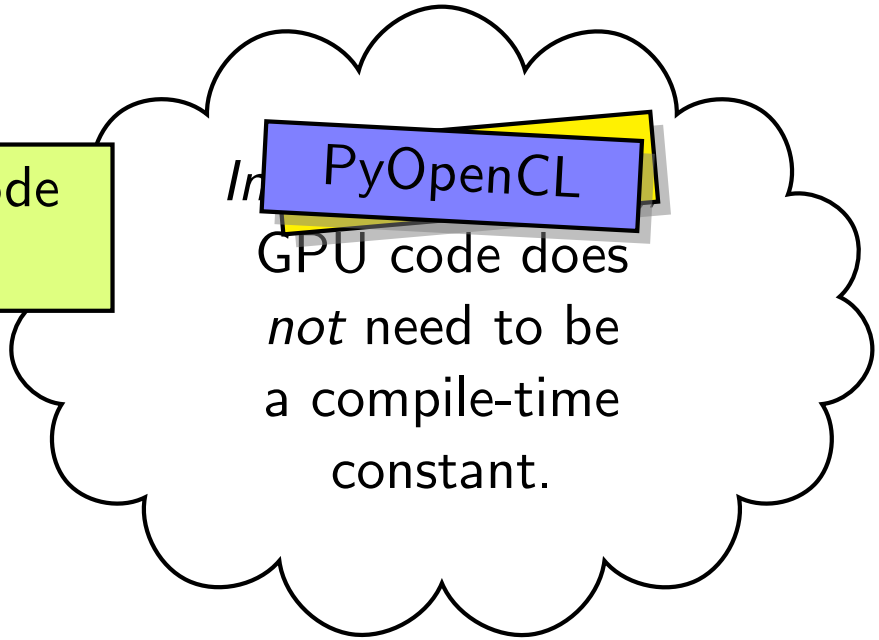
(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Metaprogramming



Good for code generation



(Key: Code is data–it *wants* to be reasoned about at run time)

credit: Andreas Klöckner (NYU)

Machine-generated Code

Why machine-generate code?

- Automated Tuning
  (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables
  ($\rightarrow$ register pressure)
- Loop Unrolling

credit: Andreas Klöckner (NYU)

RTCG via Templates

```
from jinja2 import Template
tpl = Template("""
    __global__ void twice({{ type_name }} *tgt)
    {
      int idx = threadIdx.x +
        {{ thread_block_size }} * {{ block_size }}
        * blockIdx.x;

      {% for i in range( block_size ) %}
          {% set offset = i* thread_block_size %}
          tgt[idx + {{ offset }}] *= 2;
      {% endfor %}
    }""")

rendered_tpl = tpl.render(
    type_name="float", block_size=block_size,
    thread_block_size=thread_block_size)

smod = SourceModule(rendered_tpl)
```

credit: Andreas Klöckner (NYU)

Resources / Credits

- OpenCL: *khronos.org/opencl*
- CUDA: *nvidia.com/cuda*
- Andreas Klöckner (NYU) *slides*, software: *PyCUDA* and *PyOpenCL*
- Kirk and Hwu (UIUC) *ECE498*
- John Owens (UC Davis) *EEC 227*

thank you.

flickr/peasap - “Halloween - The Aftermath”